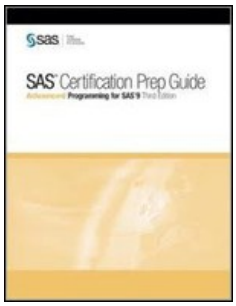


SAS Certification Prep Guide: Advanced Programming for SAS 9, Third Edition

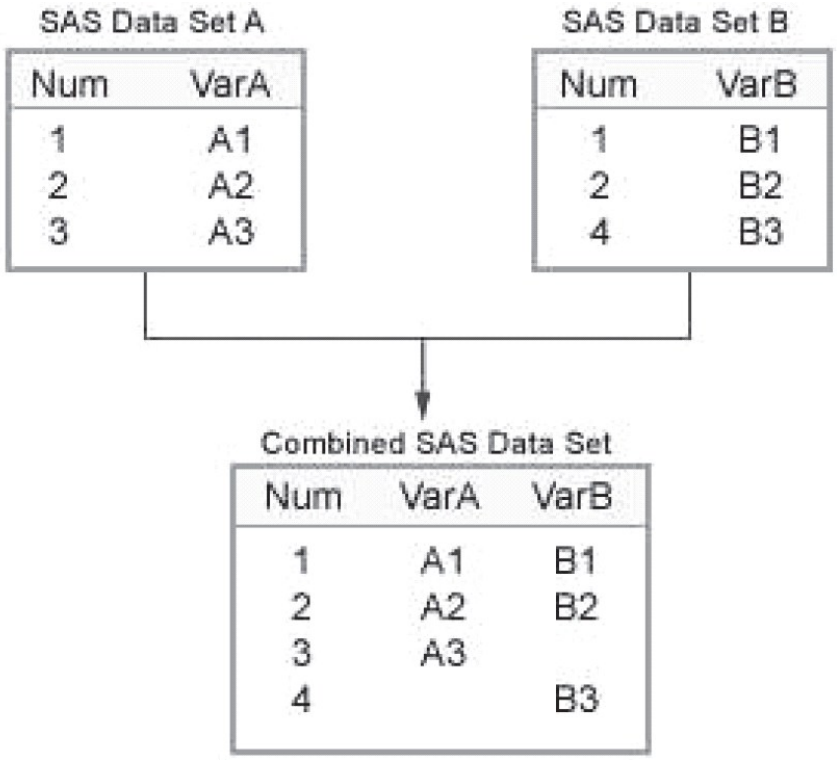
by SAS Institute

Chapter 15: Combining Data Horizontally

Overview

Introduction

Combining data horizontally refers to the process of merging or joining multiple data sets into one data set. This process is referred to as a horizontal combination because in the final data set, each observation (or horizontal row) will have variables from more than one input data set.



It is useful to combine data horizontally if you have several data sets that contain different but related information. For example, suppose you have one data set that contains employee data with the variables **`IDNumber`**, **`Name`**, and **`Address`**, and another data set that contains employee data with the variables **`IDNumber`** and **`Salary`**.

You can combine the data from these two input data sets horizontally to create an output data set that contains **`IDNumber`**, **`Name`**, **`Address`**, and **`Salary`**.

There are several methods for combining data horizontally. This chapter focuses on several methods of combining data horizontally in the DATA step, and compares a DATA step match-merge with a PROC SQL join. This chapter also covers several techniques for horizontally combining data from an input data set with values that are not stored in a SAS data set.

Objectives

In this chapter, you learn to

- identify factors that affect which technique is most appropriate for combining data horizontally
- use the IF-THEN/ELSE statement, SAS arrays, or user-defined SAS formats to combine data horizontally
- use the DATA step with the MERGE statement to combine data sets that do not have a common variable
- use the SQL procedure to combine data sets that do not have a common variable
- identify the differences between the DATA step match-merge and the PROC SQL join

- create an output data set that contains summary statistics from PROC MEANS
- combine summary statistics in a data set with a detail data set
- calculate summary data and combine it with detail data within one DATA step
- use the SET statement with the KEY= option to combine two SAS data sets
- use an index to combine two data sets
- use _**IORC**_ to determine whether an index search was successful
- use the UPDATE statement to update a master data set with a transactional data set.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

- "Performing Queries Using PROC SQL" on page 4
- "Creating Samples and Indexes" on page 470

Reviewing Terminology

Overview

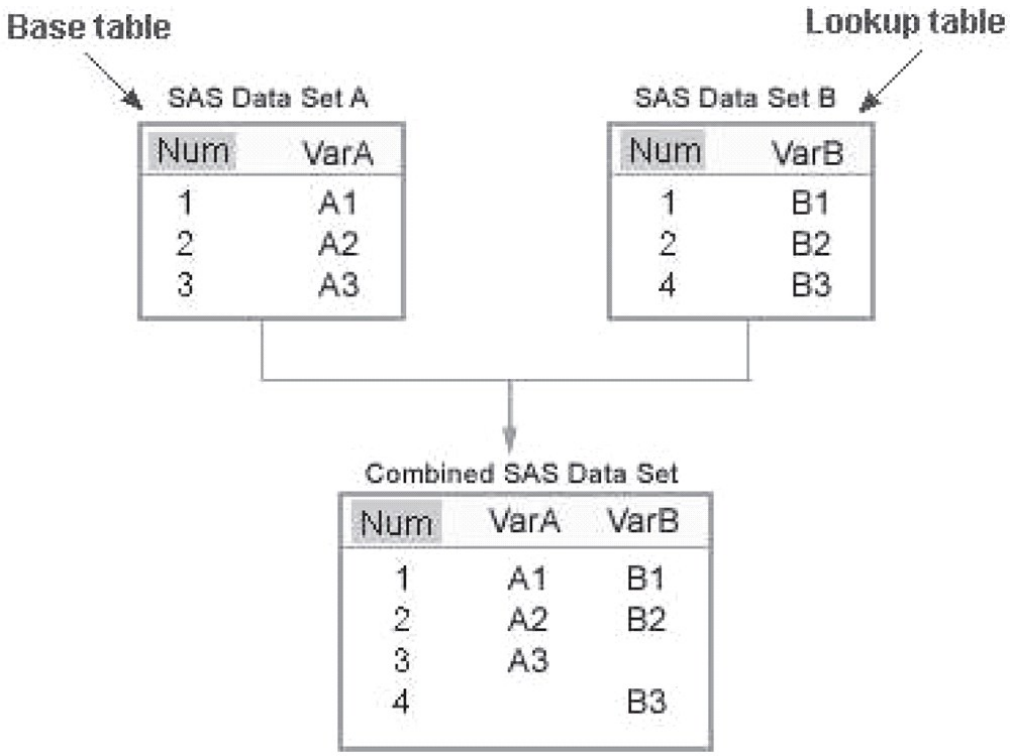
Before examining the various techniques for combining data horizontally, we will review some of the terminology that this chapter uses. The table below lists important terms that you will need to know, along with their definitions.

Term	Definition
combining data horizontally	A technique in which information is retrieved from an auxiliary source or sources, based on the values of variables in the primary source.
performing a table lookup	A technique in which information is retrieved from an auxiliary source or sources, based on the values of variables in the primary source.
base table	The primary source in a horizontal combination. In this chapter, the base table is always a SAS data set.
lookup table(s)	All input data sources, except the base table, that are used in a horizontal combination.
lookup values	The data values that are retrieved from the lookup table(s) during a horizontal combination.
key variable(s)	One or more variables that reside in both the primary file and the lookup file. The values of the key variable(s) are the common elements between the files. Often, key values are unique in the base file but are not necessarily unique in the lookup file(s).
key value(s)	For each observation, the value(s) for the key variable(s).

Note The terms *combining data horizontally* and *performing a table lookup* are synonymous and are used interchangeably throughout this chapter.

Note This chapter compares PROC SQL techniques with DATA step techniques. In PROC SQL terms, a *SAS data set* is usually referred to as a *table*, a *variable* is usually referred to as a *column*, and an *observation* is usually referred to as a *row*.

The following figure illustrates a base table and a lookup table that are used in a horizontal combination. The key variable is **`Num`**. The key values are listed vertically below **`Num`**.



Relationships between Input Data Sources

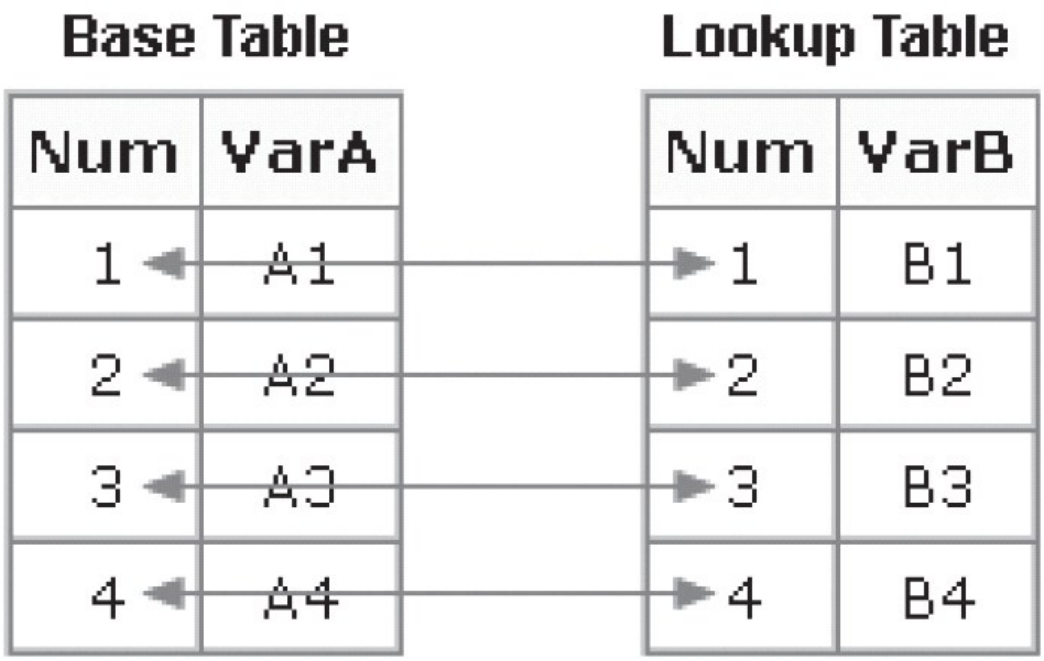
One important factor to consider when you perform a table lookup is the relationship between the input data sources. In order to combine data horizontally, you must be able to match observations from each input data source. For example, there might be one or more variables that are common to each input data source. The relationship between input data sources describes how the observations in one source relate to the observations in the other source according to these key values.

The following terms describe the possible relationships between base tables and lookup tables:

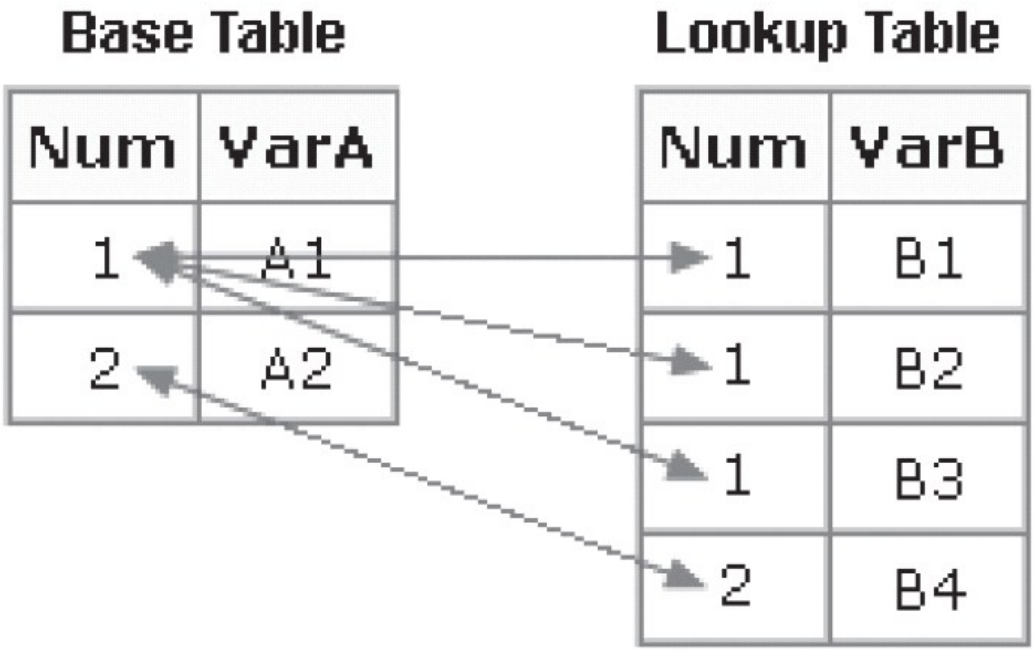
- one-to-one match
- one-to-many match
- many-to-many match
- nonmatching data.

We will look at each of these relationships in more detail.

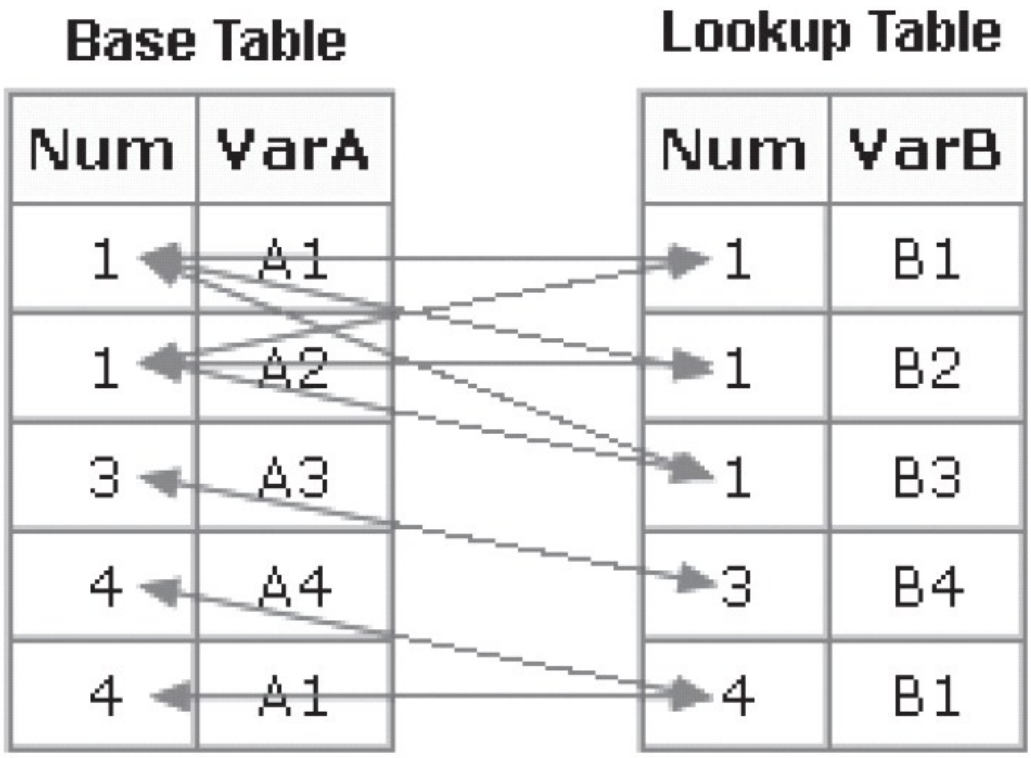
In a *one-to-one match* key values in both the base table and the lookup table are unique. Therefore, for each observation in the base table, no more than one observation in the lookup table has a matching key value.



In a *one-to-many match*, key values in the base table are unique, but key values in the lookup table are not unique. That is, for each observation in the base table there can be one observation or possibly multiple observations in the lookup table that have a matching key value.

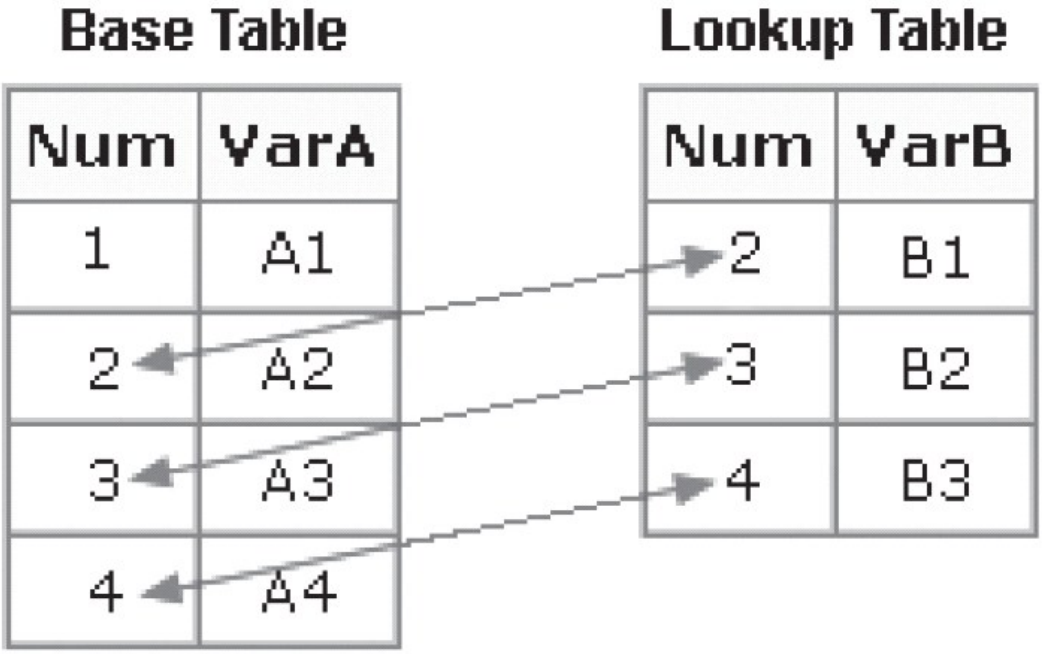


In a *many-to-many match*, key values are not unique in the base table or in the lookup table. That is, at least one observation in the base table matches multiple observations in the lookup table, and at least one observation in the lookup table matches multiple observations in the base table.



Sometimes you will have a one-to-one, a one-to-many, or a many-to-many match that also includes *nonmatching data*. That is, there are observations in the base table that do not match any observations in the lookup table, or there are observations in the lookup table that do not have matching observations in the base table. If your base table or lookup table(s) include nonmatching data, you will have one of the following:

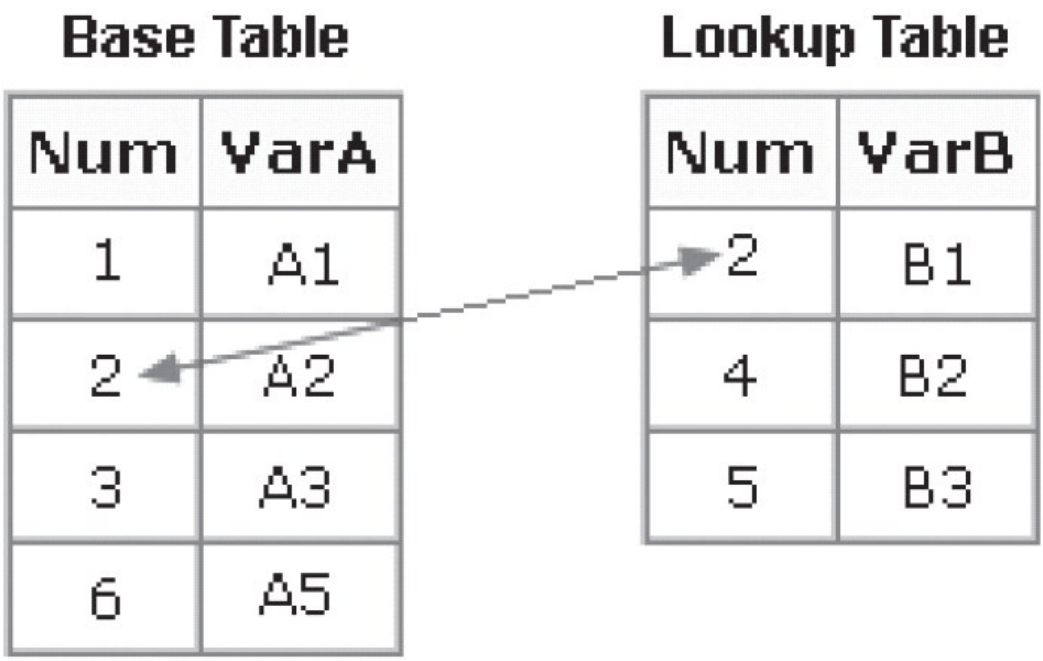
- *a dense match*, in which nearly every observation has a matching observation in the corresponding table. In the following figure, the first observation in the base table is unmatched.



Note A dense match can also refer to a relationship in which every observation has a matching observation in the corresponding table and there is no nonmatching data.

- *a sparse match*, in which there are more unmatched observations than matched observations in either the base table

or the lookup table. In the following figure, the first, third, and fourth observations in the base table, and the second and third observations in the lookup table are unmatched.



Working with Lookup Values Outside of SAS Data Sets

Overview

Remember that it is not necessary for your lookup table to be a SAS data set. Suppose you want to combine the data from your base table with lookup values that are not stored in a SAS data set. You can use the following techniques to hard-code lookup values into your program:

- the IF-THEN/ELSE statement
- SAS arrays
- user-defined SAS formats.

The IF-THEN/ELSE Statement

You should be familiar with the syntax and use of the IF-THEN/ELSE statement. Overall, this technique is easy to use and easy to understand. Because of its simplicity and because you can use other DATA step syntax with it, the IF-THEN/ELSE statement can be quite versatile as a technique for performing lookup operations. You can use this technique if your lookup values are not stored in a data set, and you can use it to handle any of the possible relationships between your base table and your lookup table if your lookup values are stored in a data set. You can use it to retrieve single or multiple values. For example, you can use DO groups to provide multiple values based on a condition.

Keep in mind that this technique will require maintenance. If you expect your lookup values to change and you have a large number of lookup values, or if you use the lookup values in multiple programs, the resources required for maintaining the IF-THEN/ELSE statements in your programs might make this technique inappropriate. Also, this technique might result in a prohibitively long program or even in a program that will not execute because it times out.

Example: Using the IF-THEN/ELSE Statement to Combine Data

Suppose you have a data set, *Mylib.Employees*, that contains information about employees. *Mylib.Employees* contains a variable named **`IDnum`** that records each employee's unique identification number. If you want to combine the data from *Mylib.Employees* with a list of employees' birthdates that is not stored in a data set, you can use the IF-THEN/ELSE statement to do so.

```
data mylib.employees_new;
   set mylib.employees;
   if IDnum=1001 then Birthdate='01JAN1963'd;
   else if IDnum=1002 then Birthdate='08AUG1946'd;
   else if IDnum=1003 then Birthdate='23MAR1950'd;
   else if IDnum=1004 then Birthdate='17JUN1973'd;
run;
```

Note *Mylib.Employees* is a fictitious data set that is used for this example and for the examples on the next two pages.

SAS Arrays

You should be familiar with the syntax and use of the ARRAY statement. With the ARRAY statement, you can either hard-code your lookup values into the program, or you can read them into the array from a data set. Elements of a SAS array are referenced positionally. That is, you use a numeric value as a pointer to the array element, so you must be able to identify elements of the array either by position or according to another numeric value. You can use multiple values or numeric mathematical expressions to determine the array element to be returned. Exact matches are not required with this technique.

The memory requirements for loading the entire array can be a drawback to using the ARRAY statement to perform a table lookup. Also, this technique is capable of returning only a single value from the lookup operation. Finally, the dimensions of the array must be supplied at compile time either by hard-coding or through the use of macro variables.

Example: Using the ARRAY Statement to Combine Data

We will consider our example of combining the data from *Mylib.Employees* with a list of lookup values. Remember that *Mylib.Employees* contains data about employees, which includes their identification numbers **(`IDnum`)** but does not include their birthdates. You can use the ARRAY statement to hard-code the birthdates into a temporary array named **`birthdates`**, and then use the array to combine the **`birthdates`** with the data in *Mylib.Employees*.

In the following DATA step, the values that are specified as subscripts for the array correspond to values of the variable **`IDnum`** in the base table, *Mylib.Employees*. The assignment statement for the new variable **`Birthdate`** retrieves a value from the **`birthdates`** array according to the current value of **`IDnum`**.

```
data mylib.employees_new;
   array birthdates{1001:1004}_temporary_ ('01JAN1963'd
         '08AUG1946'd '23MAR1950'd '17JUN1973'd);
   set mylib.employees;
   Birthdate=birthdates(IDnum);
run;
```

User-Defined SAS Formats

You should be familiar with the syntax and use of the FORMAT procedure with the VALUE statement. You can associate a format with a variable permanently by using a FORMAT or ATTRIB statement in a DATA step or PROC step that creates a SAS data set. In a DATA step, you can use a PUT statement in an assignment statement in order to use the format only while the PUT function executes. In a DATA step or PROC step, you can use the PUT function in a WHERE statement in order to use the format only during execution of the PUT function.

One advantage of using formats to combine data is that you do not have to create a new SAS data set in order to perform the lookup. Formats can be used to collapse data into categories as well as to expand data, and they can change the appearance of a report without the creation of a new variable. You can create multiple formats and use all of them in the same DATA or PROC step.

The FORMAT procedure uses a binary search (a rapid search technique) through the lookup table. Another benefit of using this technique is that maintenance is centralized; if a lookup value changes, you only have to change it in one place (in the format), and every program that uses the format will use the new value.

On the other hand, the FORMAT procedure requires the entire format to be loaded into memory for the binary search, so this technique might use more memory than others if there are a large number of lookup values.

Example: Using the FORMAT Procedure to Combine Data

Once again, suppose the data set *Mylib.Employees*contains information about employees according to their employee identification numbers **(IDnum)**, but does not contain employees' birthdates. You can use a format to combine employees' birthdates with the data that is stored in *Mylib.Employees*.

The following PROC FORMAT step uses a VALUE statement to hard-code the lookup values in the BIRTHDATE format. Then the DATA step uses the PUT function to associate the lookup values from the format with the values of **IDnum**, uses the INPUT function to associate the lookup value with the **DATE9.** informat, and assigns the formatted values to a new variable named **Birthdate**.

```
proc format;
   value birthdate 1001 = '01JAN1963'
                   1002 = '08AUG1946'
                   1003 = '23MAR1950'
                   1004 = '17JUN1973';
   run;

   data mylib.employees_new;
   set mylib.employees;
   Birthdate=input(put(IDnum,birthdate.),date9.);
run;
```

Combining Data with the DATA Step Match-Merge

The DATA Step Match-Merge

You should already know how to merge multiple data sets in the DATA step when there is a *BY variable* that is common to each of the input data sets. When you use the MERGE statement to perform a table lookup operation, your lookup values must be stored in one or more SAS data sets. Also, this technique requires that both the base table and the lookup table(s) be either sorted by or indexed on the BY variable(s).

You can specify any number of input data sets in the MERGE statement as long as all input data sets have a common BY variable. Also, the MERGE statement can combine data sets of any size. The MERGE statement is capable of returning multiple values, and you can use multiple BY variables to perform lookups that are dependent on more than one variable. The MERGE statement returns both matches and non-matches by default, but you can use DATA step syntax to return only exact matches or to include only specific values from the lookup table.

Caution Although you can use the MERGE statement to combine data from sources that have any type of relationship, this technique might *not* produce the desired results when you are working with a many-to-many match. When the data sets are merged in a DATA step, the observations are matched and combined sequentially. Once an observation is read, it is never re-read. That is, the DATA step MERGE statement does not create a Cartesian product. Therefore, the DATA step MERGE statement is probably not an appropriate technique to use for performing lookup operations when you are working with a many-to-many match.

Working with Multiple Lookup Tables

Sometimes you might need to combine data from three or more related SAS data sets in order to create one new data set. For example, the three data sets listed below all contain different information that relates to a fictional airline's flights and airports. *Sasuser.Acities* contains data about various airports, *Sasuser.Revenue* contains data about the revenue generated by various flights, and *Sasuser.Expenses* contains data about the expenses incurred by various flights. The variables in each of these data sets are listed here.

Sasuser.Acities Variables
City
Code
Country
Name

Sasuser. Revenue Variables
Date

Dest
FlightID
Origin
RevBusiness
RevEcon
Rev1st

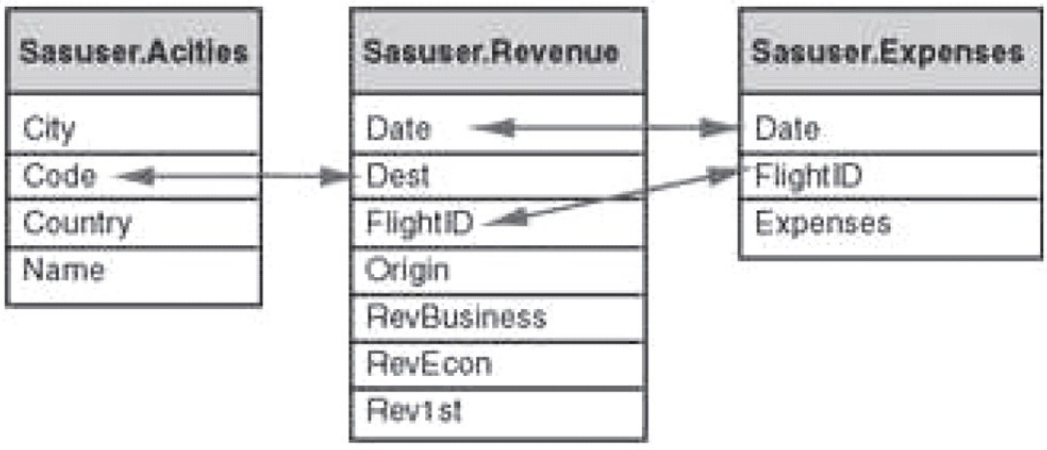
Sasuser. Expenses Variables
Date
FlightID
Expenses

Suppose you want to create a new data set, named *Sasuser.Alldata*, that contains data from each of these three input data sets. As shown below, the *Sasuser.Alldata* data set contains the variable **`Profit`**, which is calculated from the revenue values that are stored in *Sasuser.Revenue* and the expense values that are stored in *Sasuser.Expenses*.

Sasuser.All Data Variables
Date
Dest
FlightID
Origin
Profit
DestAirport
DestCity

You know that you can specify any number of input data sets in the MERGE statement as long as all input data sets have a common BY variable. However, you can see from the data set variable lists above that these three data sets do not have one common variable. We will consider a method for performing a match-merge on these three data sets.

In order to use the MERGE statement in the DATA step to combine data sets, all input data sets must have a common variable. Although the three data sets *Sasuser.Acities, Sasuser.Revenue*, and *Sasuser.Expenses* do not have a common BY variable, there are several variables that are common to *two of the three* data sets. As shown below, **`Date`** and **`FlightID`** are both common to *Revenue* and *Expenses*. The variable **`Code`** in the *Acities* data set and the variable **`Dest`** in the *Revenue* data set are also common variables.



Notice that **`Code`** in *Acities* and **`Dest`** *in Revenue* are listed as corresponding to one another even though they have different names. When you are looking for common variables between data sets, the variable names are not important

since they can be changed with the RENAME= option in the MERGE statement. Instead, you should look for variables that record the same information and that have the same type in each input data set. Common variables do not need to have the same length, although you should remember that the length of the variable in the first-listed data set will determine the length of the variable in the output data set.

Note Any variables that have the same name in multiple data sets in the MERGE statement must also have the same type. If any variables in different input data sets have identical names but do not have identical types, ERROR and WARNING messages are written to the SAS log, and the match-merge fails.

In this case, both **Code** in *Acities* and **Dest** in *Revenue* record the three-letter abbreviation of an airport.

Tip You can use PROC CONTENTS to view information about variables such as type, length, and description.

Since there are variables that are common to two different pairs of the three data sets shown above, you can combine these data sets into one data set by using the MERGE statement in two subsequent DATA steps. That is, you can perform one match-merge on two of the data sets to create one new data set that combines information from the two. Then you can perform another match-merge on the new data set and the remaining original data set. Consider the following example.

Example

In the following program, both *Sasuser.Expenses* and *Sasuser.Revenue* are sorted by **FlightID** and **Date** and are placed into temporary data sets in order to prepare them for being merged. Then these two sorted data sets are merged in a DATA step that creates a temporary output data set named *Revexpns*. In order to reduce the total number of variables in the output data set, a new variable named **Profit** is created, and the variables that are used to create **Profit** are dropped from *Revexpns*.

```
proc sort data=sasuser.expenses out=expenses;
   by flightid date;
run;

proc sort data=sasuser.revenue out=revenue;
   by flightid date;
run;

data revexpns (drop=rev1st revbusiness revecon expenses);
   merge expenses(in=e) revenue(in=r);
   by flightid date;
   if e and r;
   Profit=sum(rev1st, revbusiness, revecon, -expenses);
run;
```

Note The use of the temporary IN= variables **E** and **R** in the IF statement above ensures that only observations that contain data from each of the two input data sets are included in the output data set.

In the following program, the output data set named *Revexpns* is sorted by **Dest.** *Sasuser.Actities* is sorted by **Code** and is placed in a temporary data set. Remember that **Dest** and **Code** are corresponding variables even though they have different names.

The sorted data sets are then merged in a DATA step. Since two data sets must have at least one variable that matches exactly in order to be merged, the RENAME= option renames **Code** to **Dest** in the output data set. The DATA step merges *Revexpns* and *Acities* into a new output data set named *Alldata*.

```
proc sort data=revexpns;
   by dest;
run;

proc sort data=sasuser.acities out=acities;
   by code;
run;

data sasuser.alldata;
   merge revexpns(in=r) acities
         (in=a rename=(code=dest) keep=city name code);
   by dest;
   if r and a;
run;
```

```
proc print data=sasuser.alldata(obs=5) noobs;
   title 'Result of Merging Three Data Sets';
   format Date date9.;
run;
```

The PROC PRINT step prints the first five observations in the *Sasuser.Alldata* data set that is created in this example, as shown here.

Result of Merging Three Data Sets

FlightID	Date	Origin	Dest	Profit	City	Name
IA03300	06DEC1999	RDU	ANC	34010	Anchorage, AK	Anchorage International Airport
IA03300	18DEC1999	RDU	ANC	73471	Anchorage, AK	Anchorage International Airport
IA03300	30DEC1999	RDU	ANC	77755	Anchorage, AK	Anchorage International Airport
IA03301	13DEC1999	RDU	ANC	110402	Anchorage, AK	Anchorage International Airport
IA03301	2EDEC1999	RDU	ANC	111151	Anchorage, AK	Anchorage International Airport

Using PROC SQL to Join Data

The SQL Procedure

Another method that you can use to join data sets that do not have a common variable is the SQL procedure. You should already be familiar with using PROC SQL to create a table from the results of an inner join.

In a PROC SQL step, you can choose from each input data set only the specific variables that you want to include in the new data set, and you can return multiple values. The input data sets do not need to contain a common BY variable, nor do they need to be sorted or indexed. However, if the lookup table does have an index, the SQL procedure can take advantage of the index to provide faster retrieval of lookup values.

You can join up to 256 tables by using the SQL procedure, and you can use this technique to combine data horizontally from sources that have any type of relationship (one-to-one, one-to-many, many-to-many, or non-matching). Exact matches are returned by default from an inner join.

Note Although numerous types of joins are possible with PROC SQL, only *inner joins* are discussed in this chapter. Therefore, in the remainder of this chapter, a *PROC SQL join* refers to an inner join on multiple tables, whose results are stored in a new table. You can learn more about PROC SQL joins in "Combining Tables Horizontally Using PROC SQL" on page 86.

One drawback to using the SQL procedure to perform table lookups is that you cannot use DATA step syntax with PROC SQL, so complex business logic is difficult to incorporate into the join. However, by using PROC SQL you can often accomplish in one step what it takes multiple PROC SORT and DATA steps to accomplish.

Example: Working with Multiple Lookup Tables

The following example joins *Sasuser.Revenue, Sasuser.Expenses*, and *Sasuser.Acities* into a new data set named *Work.Sqljoin:*

```
proc sql;
   create table sqljoin as
      select revenue.flightid, revenue.date format=date9.,
             revenue.origin, revenue.dest,
             sum(revenue.rev1st,
                 revenue.revbusiness,
                 revenue.revecon)
             -expenses.expenses as Profit,
             acities.city,
             acities.name
      from sasuser.expenses, sasuser.revenue,
         sasuser.acities
      where expenses.flightid=revenue.flightid
          and expenses.date=revenue.date
          and acities.code=revenue.dest
```

```
      order by revenue.dest,
               revenue.flightid,
               revenue.date;
quit;
proc print data=work.sqljoin(obs=5);
   title 'Result of Joining Three Data Sets';
run;
```

The PROC PRINT step produces the first five observations of the *Work.Sqljoin* data set that is created in the PROC SQL step above, as shown here:

Result of Joining Three Data Sets

Obs	FlightID	Date	Origin	Dest	Profit	City	Name
1	IA03300	06DEC1999	RDU	ANC	34010	Anchorage, AK	Anchorage International Airport
2	IA03300	18DEC1999	RDU	ANC	73471	Anchorage, AK	Anchorage International Airport
3	IA03300	30DEC1999	RDU	ANC	77755	Anchorage, AK	Anchorage International Airport
4	IAQ3301	13DEC1999	RDU	ANC	110402	Anchorage, AK	Anchorage International Airport
5	IA03301	25DEC1999	RDU	ANC	111151	Anchorage, AK	Anchorage International Airport

Note Notice that the *Work.Sqljoin* data set is identical to the *Sasuser.Alldata* data set that was previously created in the DATA step merge.

Comparing DATA Step Match-Merges and PROC SQL Joins

Overview

You have seen that it is possible to create identical results with a DATA step matchmerge and a PROC SQL inner join. Although the results might be identical, these two processes are very different, and trade-offs are associated with choosing one method over the other. The following tables summarize some of the advantages and disadvantages of each of these two methods.

Table 15.1: DATA Step Match-Merge

Advantages	Disadvantages
▪ There is no limit to the number of input data sets, other than memory. ▪ Allows for complex business logic to be incorporated into the new data set by using DATA step processing, such as arrays and DO loops, in addition to MERGE features. ▪ Multiple BY variables enable lookups that depend on more than one variable.	▪ Data sets must be sorted by or indexed on the BY variable(s) before merging. ▪ The BY variable(s) must be present in all data sets, and the names of the key variable(s) must match exactly. ▪ An exact match on the key value(s) must be found.

Table 15.2: PROC SQL Join

Advantages	Disadvantages
▪ Data sets do not have to be sorted or indexed, but an index can be used to improve performance. ▪ Multiple data sets can be joined in one step without having common variables in all data sets. ▪ You can create data sets (tables), views, or query reports with the combined data.	▪ The maximum number of tables that can be joined at one time is 32. ▪ Complex business logic is difficult to incorporate into the join. ▪ PROC SQL might require more resources than the DATA step with the MERGE statement for simple joins.

Although it is possible to produce identical results with a DATA step match-merge and a PROC SQL join, these two processes will not always produce results that are identical by default. These two techniques work differently. In order to decide which technique you should use in a particular situation, you should carefully consider both the data that you want to combine and the results that you want to produce.

Consider the following simplified examples to see how each method works in various circumstances.

The following two steps show two different ways to produce the same combination of two data sets, *Data1* and *Data2*, that have a common variable, **x**. If *Data1* contains two variables, **x** and **y**, and *Data2* contains two variables, **x** and **z**, then both of the following steps produce an output data set named *Data3* that contains three variables, **x**, **y**, and **z**.

Note The code shown in the following two steps illustrates a simple comparison of a DATA step match-merge and a PROC SQL join. This comparison will be explored in the next several sections.

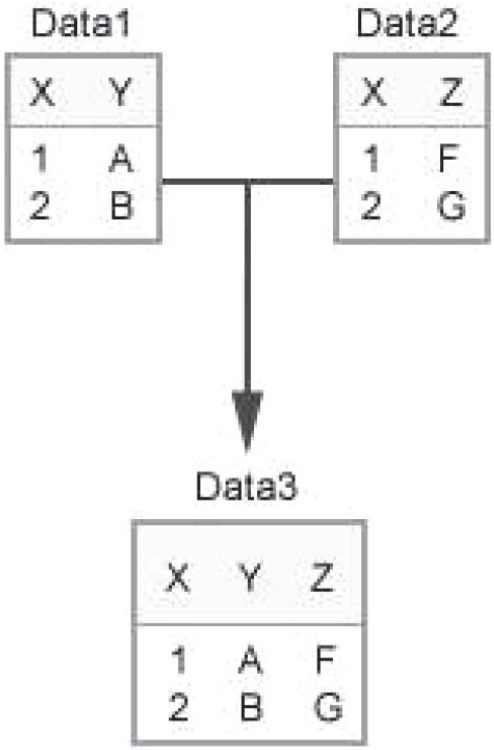
```
data data3;
   merge data1 data2;
   by x;
run;

proc sql;
   create table data3 as
      select data1.x, data1.y, data2.z
         from data1, data2
         where data1.x=data2.x;
quit;
```

The contents of *Data3* will vary depending on the values that are in each input data set and on the method used for merging. Consider the following examples.

Examples

One-to-one matches produce *identical results* whether the data sets are merged in a DATA step or joined in a PROC SQL step. Suppose that *Data1* and *Data2* contain the same number of observations. Also, suppose that in each data set, the values of **x** are unique, and that each value appears in both data sets.

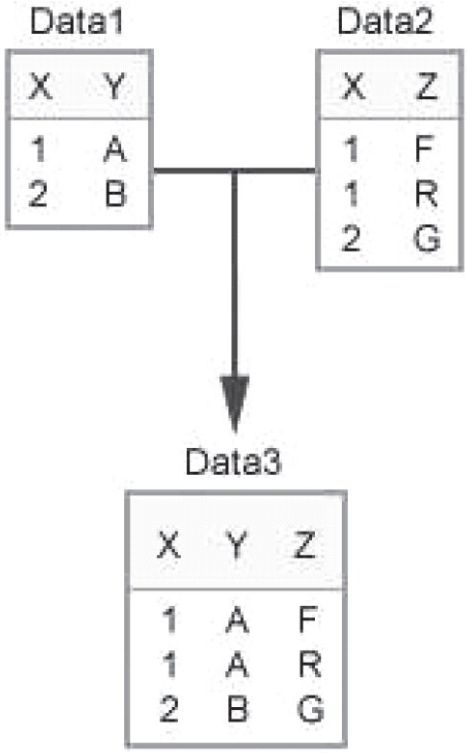
When these data sets are either merged in a DATA step or joined in a PROC SQL step, *Data3* will contain one observation for each unique value of **x**, and it will have the same number of observations as *Data1* and *Data2*.



One-to-many matches produce *identical results* whether the data sets are merged in a DATA step or joined in a PROC SQL step. Suppose that *Data1* contains unique values for **x**, but that *Data2* does not contain unique values for **x**. That is, *Data2* contains multiple observations that have the same value of **x** and therefore contains more observations than *Data1*.

When these two data sets are either merged in a DATA step or joined in a PROC SQL step, *Data3* will contain the same

number of observations as *Data2*. In *Data3*, one observation from *Data1* that has a particular value for **x** might be matched with multiple observations from *Data2* that have the same value for **x.**

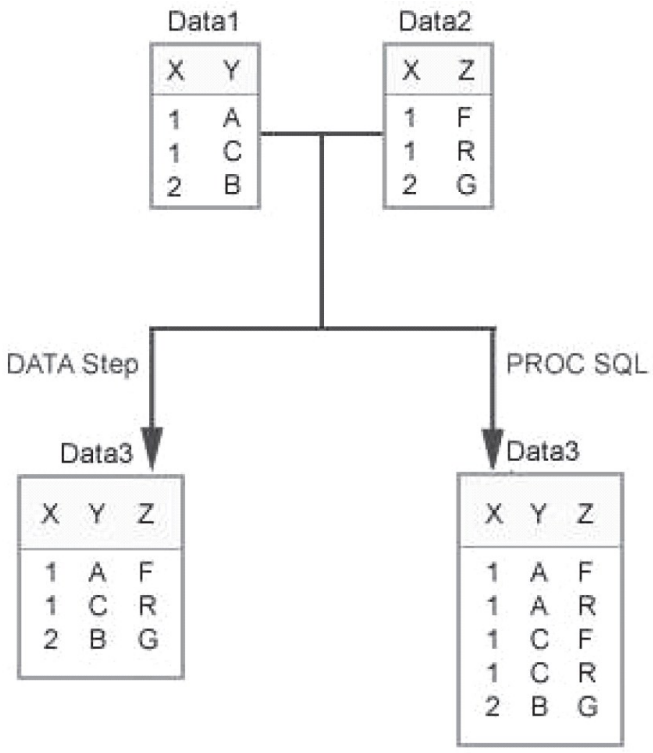


Many-to-many matches produce *different results* depending on whether the data sets are merged in a DATA step or joined in a PROC SQL step. Suppose the values of **x** are not unique in either *Data1* or in *Data2*.

When the data sets are merged in a DATA step, the observations are matched and combined sequentially. That is, an observation from the first input data set will be combined with the first observation from the second input data set that has a matching value for the BY variable. Although there might be additional observations in the second input data set that match the observation from the first input data set, these will not be included in the output data set.

In the example below, *Data3* will contain the same number of observations as the larger of the two input sets. In cases where there is a many-to-many match on the values of the BY variable, a DATA step match-merge probably does not produce the desired output because the output data set will not contain all of the possible combinations of matching observations.

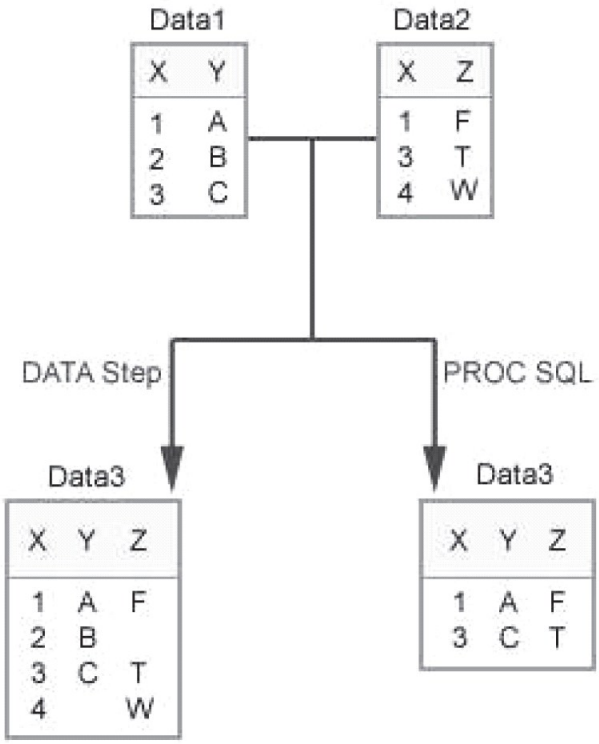
When the data sets are joined in a PROC SQL step, each match appears as a separate observation in the output data set. In the example below, the first observation that has a value of 1 for **x** in *Data1* is matched and combined with each observation from *Data2* that has a value of 1 for **x.** Then, the second observation that has a value of 1 for **x** in *Datalis* matched and combined with each observation from *Data2* that has a value of 1 for **x**, and so on.



Nonmatching data between the data sets produces *different results* depending on whether the data sets are merged in a DATA step or combined by using a PROC SQL inner join.

When data sets that contain nonmatching values for the BY variable are merged in a DATA step, the observations in each are processed sequentially. *Data3* will contain one observation for each unique value of **x** that appears in either *Data1* or *Data2*. For those values that have a nonmatching value for **x**, the observation in *Data3* will have a missing value for the variable that is taken from the other input data set.

When data sets that contain nonmatching values for the BY variable are joined in a PROC SQL step, the output data set will contain only those observations that have matching values for the BY variable. In the example below, *Data3* does not have any observations with missing values, because any observation from *Data1* or from *Data2* that contains a nonmatching value for **x** is not included in *Data3*.



You have seen the results of DATA step match-merges and PROC SQL joins in several simple scenarios. To help you understand the differences more fully, consider how the DATA step processes a match-merge and at how PROC SQL processes a join.

DATA Step Match-Merge

When you merge data sets in a DATA step, the observations in each input data set are read sequentially and are matched and combined in the output data set. The example below depicts a DATA step match-merge of two simple input data sets.

Execution of a DATA Step Match-Merge

1. This example shows the execution of the DATA step below. This DATA step creates a new data set by performing a basic match-merge on two input data sets.

```
data work.data3;
   merge data1 data2;
   by x;
run;
```

2. During the compilation phase, SAS reads the descriptor portions of the input data sets and creates the PDV. Also, SAS determines the BY groups in each input data set for the variables that are listed in the BY statement.

3. Execution begins. SAS looks at the first BY group in each input data set to determine whether the BY values match. If so, SAS reads the first observation of that BY group from the first input data set and records the values in the PDV.

4. Because the BY values match, SAS reads the first observation of the same BY group from the second input data set and records the remaining values in the PDV.

5. SAS writes the merged data to the output data set.

6. SAS continues to merge observations in the same manner until it has written all observations from the first BY group to the new data set. In this example, there is one observation in the new data set that results from the first BY group (**x** = 1).

7. If the BY values do not match, SAS reads the input data set with the lowest BY value. If the BY values match the BY values from the previous observations, SAS does not reinitialize the PDV and overwrites the values in the PDV. If the values do not match the previous BY values, SAS reinitializes the PDV. The PDV and the output data set then contain

missing values for variables that are unique to the other data set.

8. If an input data set does not contain any observations in a particular BY group, the PDV and the output data set contain missing values for the variables that are unique to that input data set.

9. SAS continues to match-merge observations until all observations from both input data sets have been read and written to the new data set. In this example, **`Work.Data3`** contains three variables and four observations.

PROC SQL Join

A PROC SQL join uses a different process than a DATA step merge to combine data sets.

Conceptually, PROC SQL first creates a *Cartesian product* of all input sets. That is, PROC SQL first matches *each observation with every other observation* in the other input data sets. Then, PROC SQL eliminates any observations from the result set that do not satisfy the WHERE clause of the join statement. The PROC SQL query optimizer contains methods to minimize the Cartesian product that must be built.

Execution of a PROC SQL Join

1. This example shows the execution of the PROC SQL step below. This PROC SQL step creates a new data set to hold the results of an inner join on two input data sets. This animation provides a conceptual view of how PROC SQL works rather than a literal depiction of the join process. In reality, SAS uses optimization routines that make the process more efficient.

```
proc sql;
   create table work.data4 as
      select *
        from datai, data 2
        where data1.x=data2.x;
quit;
```

2. Conceptually, PROC SQL first creates a Cartesian product of the two input data sets, where each observation from the first data set is combined with each observation from the second data set. PROC SQL starts by taking the first observation from *Work.Data1* and combining it with the first observation of *Work.Data2*.

3. Next, PROC SQL takes the first observation from *Work.Data1* and combines it with the second observation from *Work.Data2*.

4. PROC SQL continues in this manner until it has combined each observation from *Work.Data1* with every observation from *Work.Data2*. This is the Cartesian product of the two input data sets.

5. Finally, PROC SQL eliminates from the output data set those observations that do not satisfy the condition in the WHERE clause of the program. In this example, observations that do not have matching values for **`x`** are eliminated so that the two columns for **`x`** have identical values for each observations.

6. The results are written to the output data set. In SAS data sets, column names must be unique. Only one column **`x`** is in the output data set *Work.Data4*. In this example, *Work.Data4* contains three variables and four observations. None of the observations in *Work.Data4* contains any missing values.

Earlier in this chapter, you learned that a DATA step match-merge will probably not produce the desired results when the data sources that you want to combine have a many-to-many match. You also learned that PROC SQL and the DATA step match-merge do not, by default, produce the same results when you are combining data sources that contain nonmatching data. Now that you have seen how DATA step match-merges and PROC SQL joins work, consider an example of using each of these techniques to combine data from a many-to-many match that also contains nonmatching data.

Example: Combining Data from a Many-to-Many Match

Suppose you want to combine the data from *Sasuser.Flightschedule* and *Sasuser.Flightattendants*. The *Sasuser.Flightschedule* data set contains information about flights that have been scheduled for a fictional airline. The data set *Sasuser.Flightattendants* contains information about the flight attendants of a fictional airline. A partial listing of each of these data sets is shown below.

Table 15.3: SASuser.Flightschedule (Partial

Listing)

Date	Destination	FlightNumber	EmpID
01MAR2000	YYZ	132	1739
01MAR2000	YYZ	132	1478
01MAR2000	YYZ	132	1130
01MAR2000	YYZ	132	1390
01MAR2000	YYZ	132	1983
01MAR2000	YYZ	132	1111
01MAR2000	YYZ	182	1076
01MAR2000	YYZ	182	1118

Table 15.4: Sasuser.Flightattendants (Partial Listing)

EmpID	JobCode	LastName	FirstName
1350	FA3	Arthur	Barbara
1574	FA2	Cahill	Marshall
1437	FA3	Carter	Dorothy
1988	FA3	Dean	Sharon
1983	FA2	Dunlap	Donna
1125	FA2	Eaton	Alicia
1475	FA1	Fields	Diana
1422	FA1	Fletcher	Marie

Suppose you want to combine all variables from the *Sasuser.Flightschedule* data set with the first and last names of each flight attendant who is scheduled to work on each flight. *Sasuser.Flightschedule*ta for 45 flights, and three flight attendants are scheduled to be on each flight. Therefore, your output data set should contain 135 observations (three for each flight).

You could use the following PROC SQL step to combine *Sasuser.Flightschedule* with *Sasuser.Flightattendants*.

```
proc sql;
   create table flightemps as
      select flightschedule.*, firstname, lastname
         from sasuser.flightschedule, sasuser.flightattendants
            where flightschedule.empid=flightattendants.empid;
quit;
```

The resulting *Flightemps* data set contains 135 observations.

Now, suppose you use the following DATA step match-merge to combine these two data sets.

```
proc sort data=sasuser.flightattendants out=fa;
   by empid;
run;

proc sort data=sasuser.flightschedule out=fs;
   by empid;
run;

data flightemps2;
   merge fa fs;
by empid;
run;
```

The resulting *Flightemps2* data set contains 272 observations. The DATA step match-merge does not produce the correct results because it combines the data sequentially. In the correct results, there are three observations for each unique flight from *Sasuser.Flightschedule*, and there are no missing values in any of the observations. By contrast, the results from the DATA step match-merge contain six observations for each unique flight and many observations that have missing values.

In the last example, data from two data sets that have a many-to-many match was combined. The PROC SQL join produced the correct results, but the DATA step match-merge did not. However, you can produce the correct results in a DATA step. First, consider using multiple SET statements to combine data.

Using Multiple SET Statements

You can use multiple SET statements to combine observations from several SAS data sets.

For example, the following DATA step creates a new data set named *Combine*. Each observation in *Combine* contains data from one observation in *Dataset1* and data from one observation in *Dataset2*.

```
data combine;
   set dataset1;
   set dataset2;
run;
```

When you use multiple SET statements

- processing stops when SAS encounters the end-of-file (EOF) marker on *either* data set (even if there is more data in the other data set). Therefore, the output data set contains the same number of observations as the smallest input data set.
- the variables in the program data vector (PDV) are *not* reinitialized when a second SET statement is executed.
- for any variables that are common to both input data sets, the value or values from the data set in the second SET statement will overwrite the value or values from the data set in the first SET statement in the PDV.

Keep in mind that using multiple SET statements to combine data from multiple input sources that do not have a one-to-one match can be complicated. By default, the first observation from each data set is combined, the second observation from each data set is combined, and so on, until the first EOF marker is reached in one of the data sets. Therefore, if you are working with data sources that do not have a one-to-one match, or that contain nonmatching data, you will need to add additional DATA step syntax in order to produce the results that you want.

Example: Using Multiple SET Statements with a Many-to-Many Match

Remember that in the previous example you wanted to combine *Sasuser.Flightschedule* with *Sasuser.Flightattendants*. Your resulting data set should contain all variables from the *Sasuser.Flightschedule* data set with the first and last names of each flight attendant who is scheduled to work on each flight. *Sasuser.Flightschedule* contains data for 45 flights, and three flight attendants are scheduled to be on each flight. Therefore, your output data set should contain 135 observations (three for each flight).

You can use the following DATA step to perform this table lookup operation. In this program, the first SET statement reads an observation from the *Sasuser.Flightschedule* data set. Then the DO loop executes, and the second SET statement reads each observation in *Sasuser.Flightattendants*. The **EmpID** variable in *Sasuser.Flightattendants* is renamed so that it does not overwrite the value for **EmpID** that has been read from *Sasuser.Flightschedule*. Instead, these two values are used for comparison to control which observations from *Sasuser.Flightattendants* should be included in the output data set for each observation from *Sasuser.Flightschedule*.

```
data flightemps3(drop=empnum jobcode);
   set sasuser.flightschedule;
   do i=1 to num;
      set sasuser.flightattendants
          (rename=(empid=empnum))
          nobs=num point=i;
      if empid=empnum then output;
   end;
run;
```

The resulting *Flightemps3* data set contains 135 observations and no missing values. Keep in mind that although it is possible to use a DATA step to produce the same results that a PROC SQL join creates by default, the PROC SQL step might be much more efficient.

Combining Summary Data and Detail Data

Overview

You have seen how to combine data from multiple data sets. Suppose you want to calculate percentages based on individual values from a data set as compared to a summary statistic of the data. You need to

- create a summary statistic
- combine the summary information with the detail rows of the original data set
- calculate the percentages.

For example, the data set *Sasuser.Monthsum* has one row for every value of **SaleMon** (month and year) from 1997 to 1999. Each row contains information about the revenue generated by an airline.

Note Note that the **SaleMon** variable has a label of **Sales Month** in the

Sasuser.Monthsum data set.

Table 15.5: SAS Data Set Sasuser.Monthsum (Partial Listing)

Sales Month	RevCargo	MonthNo
JAN1997	$171,520,869.10	1
JAN1998	$238,786,807.60	1
JAN1999	$280,350,393.00	1
FEB1997	$177,671,530.40	2
FEB1998	$215,959,695.50	2
FEB1999	$253,999,924.00	2

Suppose you want to produce a report that shows what percentage of the total cargo revenue for the three-year period was generated in each month of each year. You could summarize the data to get the total revenue for cargo for the three-year period and assign that value to a new variable called **Cargosum** in a summary data set.

Table 15.6: Summary Data Set

Cargosum
$8,593,432,002.35

Then you would need to combine the summary data **(Cargosum)** with the detail data in *Sasuser.Monthsum* so that you could calculate percentages of the total cargo revenue for each month.

Table 15.7: Partial Listing of the Combined Data Set

Sales Month	RevCargo	Month No	Cargosum	PctRev
JAN1997	$171,520,869.10	1	$8,593,432,002.35	<RevCargo/Cargosum>
JAN1998	$238,786,807.60	1	$8,593,432,002.35	<RevCargo/Cargosum>
JAN1999	$280,350,393.00	1	$8,593,432,002.35	<RevCargo/Cargosum>
FEB1997	$177,671,530.40	2	$8,593,432,002.35	<RevCargo/Cargosum>
FEB1998	$215,959,695.50	2	$8,593,432,002.35	<RevCargo/Cargosum>
FEB1999	$253,999,924.00	2	$8,593,432,002.35	<RevCargo/Cargosum>

We will examine this task more closely.

The MEANS Procedure

You should already know how to use the MEANS procedure for producing summary statistics. By default, PROC MEANS

generates a report that contains descriptive statistics. The descriptive statistics can be routed to a SAS data set by using an *OUTPUT statement* and the default report can be suppressed by using the *NOPRINT option*.

General form, PROC MEANS with OUTPUT statement:

```
PROC MEANS DATA=original-SAS-data-set NOPRINT;
     <VARvariable(s);>
     OUTPUT OUT= output-SAS-data-set
             statistic=output-variable(s);
RUN;
```

where

original-SAS-data-set

identifies the data set on which the summary statistic is generated.

variable(s)

is the name(s) of the variable(s) that is being analyzed.

output-SAS-data-set

names the data set where the descriptive statistics will be stored.

statistic

is one of the summary statistics generated.

output-variable(s)

names the variable(s) in which to store the value(s) *statistic* in the output data set.

The output data set that a PROC MEANS step creates contains the requested statistics as values for *output-variable(s)*, as well as two additional variables that are automatically included, as follows:

- **_TYPE_** contains information about the class variables
- **_FREQ_** contains the number of observations that an output level represents.

Example

The following program creates a summary data set named *Sasuser.Summary.Summary* contains the sum of the values of **Revcargo** from *Sasuser.Monthsum*, stored in the variable **Cargosum.**

```
proc means data=sasuser.monthsum noprint;
   var revcargo;
   output out=sasuser.summary sum=Cargosum;
run;

proc print data=sasuser.summary;
run;
```

Because of the NOPRINT option, the PROC MEANS step does not produce any output. Printing the *Sasuser.Summary* data set produces the following output.

Obs	_TYPE_	_FREQ_	Cargosum
1	0	36	$8593432002.35

Once you have created the summary statistic, you need to combine this summary information with the detail rows of the data set so that you can calculate the percentages. Remember that you can use multiple SET statements to combine data horizontally.

Consider how this process works by using multiple set statements to combine the detail rows of *Sasuser.Monthsum* with the detail data that we created in *Sasuser.Summary*.

Example

This example creates a new data set named *Percent1* that combines

- summary data (total revenue for cargo from the three-year period) from *Sasuser.Summary*
- detail data (month and total cargo for the month) from *Sasuser.Monthsum*.

Percent1 also contains a new variable named **PctRev** that records the calculated percentage of the total revenue that each observation represents.

Remember, the automatic variable **_N_** keeps track of how many times the DATA step has begun to execute. The following DATA step uses **_N_** to keep SAS from reaching the EOF marker for *Sasuser.Summary* after the first iteration of the step. Since the variables in the PDV will not be reinitialized on each iteration, the first value of *Summary.Cargosum* will be retained in the PDV for each observation that is read from *Sasuser.Monthsum*.

1. This example shows the compilation and execution of the DATA step below. This DATA step creates a new data set that combines summary data from one input data set *(Sasuser.Summary)* and detail data from a second input data set *(Sasuser.Monthsum)*.

   ```
   data sasuser.percent1(drop=cargosum);
      if _N_=1 then set sasuser.summary(keep=cargosum);
      set sasuser.monthsum(keep=salemon revcargo);
      PctRev=revcargo/cargosum;
   run;
   ```

2. During the compilation phase, SAS reads the descriptor portion of the input data set and creates the PDV. **_N_** is a temporary variable that is included in the PDV although it will not be included in the output data set.
3. Execution begins. On the first iteration of the DATA step, **_N_** has a value of 1. The IF statement evaluates as true, so the first SET statement executes and SAS reads the value of **Cargosum** from *Sasuser.Summary* and records it in the PDV.
4. Next, the second SET statement executes. SAS reads the first observation in *Sasuser.Monthsum* and records the values in the PDV.
5. SAS calculates the value of **PctRev** and records it in the PDV.
6. At the end of the DATA step, SAS write the values in the PDV to the output data set. **_N_** is not included in the output data set since it is a temporary variable. **CargoSum** is dropped from the output data set as well.
7. On the second iteration of the DATA step, the value of **_N_** is *2*, so the IF statement evaluates to false and the first SET statement does not execute. However, the value of **CargoSum** is retained in the PDV.
8. The second SET statement executes. SAS reads the second observation from *Sasuser.Monthsum* and records the values in the PDV.
9. The value for **PctRev** is calculated and recorded in the PDV. SAS will write the values in the PDV to the output data set (except for **_N_** and **CargoSum).**
10. The DATA step will continue to execute until all observations have been read from *Sasuser.Monthsum*.

Another method of combining summary data and detail data is to create the summary statistic in a DATA step and combine it with the detail data in the same step. To do this you must

- read the data once and calculate the summary statistic
- re-read the data to combine the summary statistic with the detail data and calculate the percentages.

The Sum Statement

You can use the sum statement to obtain a summary statistic within a DATA step. The sum statement adds the result of an expression to an *accumulator variable*.

General form, sum statement:

variable+expression;

where

variable

specifies the name of the accumulator variable. This variable must be numeric. The variable is automatically set to 0 before the first observation is read. The variable's value is retained from one DATA step execution to the next.

expression

is any valid SAS expression.

Caution If the *expression* produces a missing value, the sum statement ignores it. (Remember, however, that assignment statements assign a missing value if the *expression* produces a missing value.)

Note The sum statement is one of the few SAS statements that does not begin with a keyword.

The sum statement adds the result of the expression that is on the right side of the plus sign (+) to the numeric variable that is on the left side of the plus sign. At the top of the DATA step, the value of the numeric variable is not set to missing as it usually is when reading raw data. Instead, the variable retains the new value in the program data vector for use in processing the next observation.

Example

The following example uses a sum statement to generate the summary statistic in a DO UNTIL loop. On the first execution of the DATA step, the DO UNTIL loop reads each observation of *Sasuser.Monthsum* and keeps a running tally of the total value of **`RevCargo`** from each observation. On each subsequent execution of the DATA step, this tally (stored in the variable **`TotalRev)`** is divided into **`RevCargo`** in order to calculate the new variable **`PctRev.`**

Note Remember that the END= data set option creates a temporary variable that contains an end-of-file indicator.

1. This example shows the execution of the DATA step below. This DATA step reads the same data set, *Sasuser.Monthsum*, twice: first, to create a summary statistic; second, to merge the summary statistic back into the detail data to create a new data set, *Sasuser.Percent2*.

```
data sasuser.percent2(drop=totalrev);
   if _N_=1 then do until (LastObs);
      set sasuser.monthsum(keep=revcargo) end=lastobs;
      TotalRev+revcargo;
   end;
   set sasuser.monthsum(keep=salemon revcargo);
   PctRev=revcargo/totalrev;
run;
```

2. During the compilation phase, SAS reads the descriptor portion of the input data set and creates the PDV. **`_N_`**, LastObs, and TotalRev are temporary variables that are included in the PDV although they will not be included in the output data set.

3. Execution begins. The temporary variables are initialized with values. The IF statement resolves to true on the first iteration of the DATA step, so the DO UNTIL loop begins to execute. Remember, in a DO UNTIL loop, the condition is evaluated at the bottom of the loop.

4. SAS reads the first observation from *Sasuser.Monthsum* and writes the value for RevCargo to the PDV.

5. The value of **`TotalRev`** is increased by the value of **`RevCargo`** and recorded in the PDV.

6. At the bottom of the DO loop, SAS evaluates the UNTIL expression. It resolves to false since the value of **`LastObs`** is *0*, so the loop continues to execute.

7. SAS reads the second observation from *Sasuser.Monthsum*, overwriting the value for **RevCargo** in the PDV and adding this value to the accumulator variable **TotalRev.**
8. The DO UNTIL loop continues to execute until SAS reads the last observation from *Sasuser.Monthsum* and the value of **LastObs** is set to *1*. At this point, the value for **TotalRev** in the PDV is the sum of all values for **RevCargo** in *Sasuser.Monthsum*. The loop is satisfied.
9. The second SET statement reads data from the same data set as the first SET statement did. However, this time values for both **SaleMon** and **RevCargo** are recorded in the PDV. There is already a value for **TotalRev** in the PDV.
10. A value for **PctRev** is calculated for the observation and recorded in the PDV. Then, SAS writes the values in the PDV to the output data set *Sasuser.Percent2*, except for the temporary variables and the variable **TotalRev.**
11. On the second iteration of the DATA step, the value of **_N_** increases to *2*, so the IF expression is false. The second SET statement executes and values from the second observation of *Sasuser.Monthsum* are read into the PDV.
12. The value for the accumulator variable **TotalRev** is retained from the last iteration and is used to calculate a new value for **PctRev**, which is recorded in the PDV. SAS writes the values in the PDV to the output data set and the DATA step iterates.
13. The DATA step iterates until SAS has read the last observation from *Sasuser.Monthsum* and has written the new observation to the output data set *Sasuser.Percent2*.

Using an Index to Combine Data

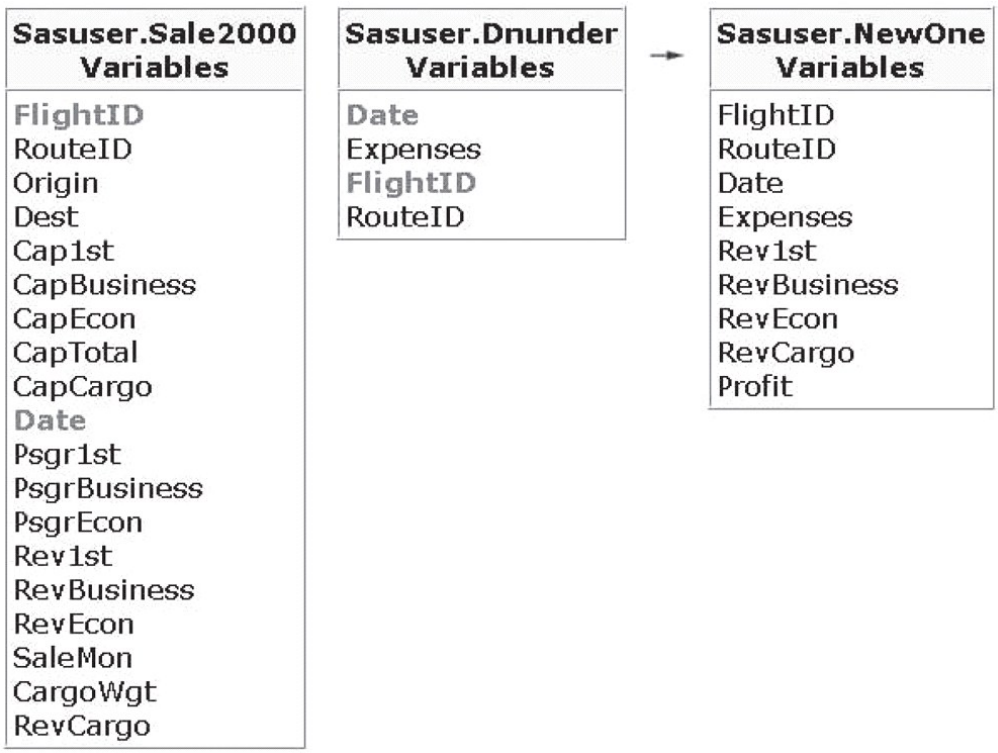
Overview

Suppose you want to combine data from two data sets, and one of the data sets is much larger than the other. Also, suppose you want to use only those observations from the larger data set that match an observation from the smaller data set according to the value of one or more common variables.

You should already know how to create an index on a SAS data set. You have learned that PROC SQL can take advantage of an index to improve performance on ajoin. You can also take advantage of an index in a DATA step to combine data from matching observations in multiple data sets if the index is built on variables that are common to all input data sets.

For example, suppose you want to combine data from the matching observations in *Sasuser.Dnunder* and *Sasuser.Sale2000*. Only a portion of the flights that are in *Sasuser.Sale2000* (which has 156 observations) are also in *Sasuser.Dnunder* (which has only 57 observations). Suppose you want to use only data from *Sasuser.Sale2000* about flights that are in both data sets.

Assume that *Sasuser.Sale2000* has a composite index named *Flightdate* associated with it. The values for *Flightdate* are unique and are based on the values of the variables **FlightID** and **Date.** You can use *Flightdate* to combine data from only the observations in both data sets that have matching values for **FlightID** and **Date.**



The next few sections show how to use the *Flightdate* index to combine matching observations from the *Sasuser.Sale2000* data set and the *Sasuser.Dnunder* data set.

The KEY= Option

You have seen how to use multiple SET statements in a DATA step in order to combine summary data and detail data in a new data set. You can also use multiple SET statements to combine data from multiple data sets if you want to combine only data from observations that have matching values for particular variables.

You specify the KEY= option in the SET statement to use an index to retrieve observations from the input data set that have key values equal to the key variable value that is currently in the program data vector (PDV).

General form, SET statement with KEY= option:

SET *SAS-data-set-name* **KEY=** *index-name;*

where

index-name

is the name of an index that is associated with the *SAS-data-set-name* data set.

To use the SET statement with the KEY= option to perform a lookup operation, your lookup values must be stored in a SAS data set that has an index. This technique is appropriate only when you are working with one-to-one matches, and you can use it with a lookup table of any size. It is possible to return multiple values with this technique, and you can use other DATA step syntax with it as well.

When SAS encounters the SET statement that includes the KEY= option, there must already be a value in the PDV for the value or values of the key variable(s) on which the KEY= index is built. SAS can then use the index to retrieve an observation that has a value for the key variable that matches the key value from the PDV.

For example, if the *Sasuser.Sale2000* data set has an index named *Flightdate* associated with it, the following SET statement uses the *Flightdate* index to locate observations in *Sale2000* that have specific values for **`FlightID`** and **`Date:`**

```
set sasuser.sale20 00 key=flightdate;
```

When the SET statement in the example above begins to execute, there must already be a value for **FlightID** and a value for **Date** in the PDV. SAS then uses the *Flightdate* index to retrieve an observation from *Sasuser.Sale2000* that has values for **FlightID** and **Date** that match the values for **FlightID** and **Date** that are already in the PDV.

In order to assign a key value in the PDV before the SET statement with the KEY= option executes, you precede that SET statement with another SET statement in the DATA step. Consider this example in context.

Example

Remember that you want to combine *Sasuser.Sale2000* and *Sasuser.Dnunder*, and that *Sasuser.Sale2000* has an index named *Flightdate* that is based on the values of the **FlightID** and the **Date** variables. You can use two SET statements to combine these two data sets, and you can use the KEY= option on the second SET statement to take advantage of the index.

In the following example,

- the first SET statement reads an observation sequentially from the *Sasuser.Dnunder* data set. SAS writes the values from this observation to the PDV, and then moves to the second SET statement.
- SAS will use the *Flightdate* index on *Sasuser.Sale2000* to find an observation in *Sasuser.Sale2000* that has values for **FlightID** and **Date** that match the values of **FlightID** and **Date** that are currently in the PDV.
- *Work.Profit* is the output data set.

Caution If you use the KEY= option to read a SAS data set, you cannot use WHERE processing on that data set in the same DATA step.

1. This example shows the execution of a DATA step that uses two SET statements to combine data from two input data sets *(Sasuser.Sale2000* and *Sasuser.Dnunder)* into one output data set *{Work.Profit)*. The DATA step also uses an index on the larger of the two input data sets, which is *Sasuser.Sale2000*, to find matching observations.

```
data work.profit;
   set sasuser.dnunder;
   set sasuser.sale2000(keep=routeid flightid date rev1st
                        revbusiness revecon revcargo)
                        key=flightdate;
   Profit=sum(rev1st, revbusiness, revecon, revcargo,
              -expenses);
run;
```

2. SAS sets up the new data set by reading the descriptor portions of the input data sets and creating the PDV.
3. The first SET statement executes. SAS reads the first observation in *Sasuser.Dnunder* and records the values in the PDV.
4. When the second SET statement executes, the KEY= option causes SAS to use the *Flightdate* index to directly access the observation in *Sasuser.Sale2000* that has values for **FlightID** and **Date** that match the values already in the PDV. SAS reads the observation and records the values to the PDV.
5. SAS calculates the value for **Profit** and records it in the PDV. Then, SAS writes the values from the PDV to the output data
6. The DATA step continues to iterate. Only the variable **Profit** is reinitialized to missing. SAS reads the second observation in *Sasuser.Dnunder* and records the values in the PDV, overwriting the values that have been retained.
7. SAS uses the *Flightdate* index to find a matching observation in *Sasuser.Sale2000*. Then, SAS records the values from that observation in the PDV, overwriting the values that have been retained. A new value for **Profit** is calculated and recorded, and the values are written to the output data set.
8. The DATA step continues to iterate until all observations have been read from *Sasuser.Dnunder*.

Remember that when SAS encounters the SET statement that includes the KEY= option, there must already be a value in the PDV for the value or values of the key variable(s) on which the KEY= index is built. Otherwise, the step will generate

errors in the output data set.

Example

The following step is identical to the last example except that the order of the SET statements has been reversed:

```
data work.profit2;
   set sasuser.sale2 0 00(keep=routeid flightid date
       rev1st revbusiness revecon revcargo)
       key=flightdate;
   set sasuser.dnunder;
   Profit=sum(rev1st, revbusiness, revecon, revcargo,
          -expenses);
run;
```

On the first iteration of this DATA step, there are no values for the key values in the PDV when SAS encounters the SET statement with the KEY= option. Therefore, SAS does not know what to look up in the index, and no observation is read from the *Sasuser.Sale2000* data set. SAS proceeds to the second SET statement, reads an observation from the *Sasuser.Dnunder* data set, and writes the values to the PDV and to the *Work.Profit2* data set.

Since no data was read from the *Sasuser.Sale2000* data set, there are missing values in the first observation of the output data set. Also, if you examine the values for **Rev1st, RevBusiness, RevEcon** and **RevCargo** in *Work.Profit2* and compare them with the values for these variables in *Work.Profit*, you will notice that there are differences between these two data sets.

Remember that the values in the PDV are not reinitialized after each iteration of the DATA step. On the second iteration of the DATA step, SAS uses the values from the first observation of *Sasuser.Dnunder* to match an observation from *Sasuser.Sale2000*. But before these values are written to the *Work.Profit2* data set, a new observation is read from *Sasuser.Dnunder* and written to the PDV. Therefore, none of the observations in *Work.Profit2* actually contains correctly matched data from the two input data sets.

You have seen how to use a SET statement with the KEY= option in conjunction with a second SET statement to create a data set that combines data from matching observations of two input data sets. Remember that when you use multiple SET statements, the variables in the PDV are *not* reinitialized when the second SET statement is executed. This can lead to problems in the output data set.

Suppose SAS reads an observation from the first input data set on the second iteration of the DATA step (that is, when **_N_**=2) and does not find a matching observation in the second input data set. Because the DATA step has already iterated once, and the values in the PDV have not been reinitialized, there are already values in the PDV for all variables. Therefore, the resulting observation in the output data set will contain values from the second observation of the first input data set, combined with values from the first observation of the second input data set.

Example

If you examine the *Work.Profit* output data set closely, you will notice that the final observation in the output data set contains values for several variables that are identical to values in the previous observation. This duplication of data is incorrect, although the error might not be obvious.

The error in the output data set is caused by a data error in one of the input data sets. If you examine the *Sasuser.Dnunder* data set closely, you will find that all of the values for **FlightID** begin with the characters *IA10* except the value in the last observation of the data set. Instead, the value for **FlightID** in the last observation begins with the characters *IA11*. This is a data error. Because of the data error, when the DATA step executes SAS will not be able to find a matching observation in *Sasuser.Sale2000* for the last observation in *Sasuser.Dnunder*, and will write an observation to the output data set that contains data from the last observation in *Sasuser.Dnunder* and data from the previous DATA step iteration for *Sasuser.Sale2000*.

The SAS log provides an additional indication that the final observation in the output data set contains nonmatching data. The observation that contains unmatched data is printed to the log. As you can see in the log sample below, the unmatched observation includes an **_Error_** variable whose value is *1*, which indicates that there is an error. The **_N_** variable indicates the iteration of the DATA step in which the error occurred.

Table 15.8: SAS Log

```
FlightID=IA11802 RouteID=0000108 Date=30DEC2000 Expenses=3720
Rev1st=1270 RevBusiness=. RevEcon=5292 RevCargo=1940 Profit=4782

_ERROR_=1_IORC_=123 0015_N_=57
NOTE: There were 57 observations read from the data set
      SASUSER.DNUNDER.
NOTE: The data set WORK.PROFIT has 57 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time           0.38 seconds
      cpu time            0.04 seconds
```

You could check the SAS log for observations that contain errors in order to ensure that your output data set does not contain bad data, but there is a better way. Notice that the observation that is printed in the SAS log above also contains a variable named **_IORC_** You can use the value of the **_IORC_** variable to prevent the observations that contain errors from being written to your output data set.

The _IORC_ Variable

When you use the KEY= option, SAS creates an automatic variable named **_IORC_**, which stands for *INPUT/OUTPUT Return Code*. You can use **_IORC_** to determine whether the index search was successful. If the value of **_IORC_** is zero, SAS found a matching observation. If the value of **_IORC_** is not zero, SAS did not find a matching observation.

Note The **_IORC_** variable is also automatically created when you use the MODIFY statement with the KEY= option in the DATA step.

In the previous section, you saw an example in which a data error was included in the output data set and was written to the SAS log. To prevent writing the data error to the log (and to your output data set)

- check the value of **_IORC_** to determine whether a match has been found
- set **_ERROR_** to *0* if there is no match
- delete the nonmatching data or write the nonmatching data to an errors data set.

Example

The following example uses the *Flighdate* index to combine data from *Sasuser.Sale2000* with data from *Sasuser.Dnunder*, and writes the combined data to a new data set named *Work.Profit3*. If any unmatched observations are read from *Sasuser.Dnunder*, the resulting combined observation will be written to *Work.Errors*. No observations should be written to the SAS log.

```
data work.profit3 work.errors;
   set sasuser.dnunder;
   set sasuser.sale2 0 00(keep=routeid flightid date rev1st
       revbusiness revecon revcargo)key=flightdate;
   if _iorc_=0 then do;
      Profit=sum(rev1st, revbusiness, revecon, revcargo,
            -expenses);
      output work.profit3;
   end;
   else do;
      _error_=0;
      output work.errors;
   end;
run;
```

If you examine the results from the program above, you will notice that there is one less observation in the *Work.Profit3* output data set that there was in the *Work.Profit* output data set. The extra observation has been written to the *Work.Errors* data set because it contains a data error.

Using a Transactional Data Set

Overview

Sometimes, rather than just combining data from two data sets, you might want to update the data in one data set with data that is stored in another data set. That is, you might want to update a master data set by overwriting certain values in it with values that are stored in a transactional data set.

For example, suppose the data set *Mylib.Empmaster* contains some information that is outdated. You have the current data stored in another data set named *Mylib.Empchanges.Mylib.Empmaster* contains 148 observations, and *Mylib.Empchanges* contains six observations. The variable **EmpID** contains unique values in both data sets.

A partial listing of *Mylib.Empmaster* and the full listing of *Mylib.Empchanges* is shown below. Notice that there is one observation in each data set with a value of *1065* for **EmpID**, and that the values of **JobCode** and **Salary** are different in this observation.

Table 15.9: Mylib.Empmaster (Partial Listing)

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
05MAR1957	30MAR1990	1009	M	TA1	$40,432
01JAN1956	20OCT1979	1017	M	TA3	$57,201
23MAY1963	27OCT1982	1036	F	TA3	$55,149
14APR1962	17SEP1990	1037	F	TA1	$39,98
13NOV1967	26NOV1989	1038	F	TA1	$37,146
17JUL1961	27AUG1984	1050	M	ME2	$49,234
29JAN1942	*10JAN1985*	*1065*	*M*	*ME2*	*$49,126*
18OCT1970	06OCT1989	1076	M	PT1	$93,181

Table 15.10: Mylib.Empchanges

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
30JUN1955	31JAN1982	1639	F	TA3	$59,164
29JAN1942	*10JAN1985*	*1065*	*M*	*ME3*	*$53,326*
03DEC1961	10Octl985	1561	M	TA3	$51,120
25SEP1965	07OCT1989	1221	F	FA3	$41,854
11AUG1970	01NOV2000	1447	F	FA1	$30,340
13SEP1968	05NOV2000	1998	M	SCP	$32,240

If you could see the full listing of *Mylib.Empmaster*, you would see that each of the observations in *Mylib.Empchanges* has a matching observation in *Mylib.Empmaster* based on the values of **EmpID.** There are also many observations in *Mylib.Empmaster* that do not have a matching observation in *Mylib.Empchanges*. To update *Mylib.Empmaster*, you want to find all of the matching observations and change their values for **JobCode** and **Salary** with the new values from *Mylib.Empchanges*. You can use the UPDATE statement to make these changes.

Using the UPDATE Statement

You use the UPDATE statement to update a master data set with a transactional data set. The UPDATE statement can perform the following tasks:

- change the values of variables in the master data set
- add observations to the master data set
- add variables to the master data set.

General form, UPDATE statement:

```
DATA master-data-set;
        UPDATE master-data-set transaction-data-set;
        BY by-variable(s);
```

```
RUN;
```

where

master-data-set

names the SAS data set used as the master file.

transaction-data-set

names the SAS data set that contains the changes to be applied to the master data set.

by-variable(s)

names a variable that appears in both *master-data-set* and in *transaction-data-set*. Each observation in *master-data-set* must have a unique value for *by-variable*, but *transaction-data-set* can contain more than one observation with the same *by-variable* value.

The UPDATE statement replaces values in the master data set with values from the transactional data set for each observation with a matching value of the BY variable. Any observations in either the master data set or the transactional data set that have non-matching values for the BY variable are included in the output data set. Also, by default, SAS does not replace existing values in the master data set with missing values if those values are coded as periods (for numeric variables) or blanks (for character variables) in the transaction data set.

When you use the UPDATE statement, keep in mind the following restrictions.

- Only two data set names can appear in the UPDATE statement.
- The master data set must be listed first.
- A BY statement that gives the matching variable must be used.
- Both data sets must be sorted by or have indexes based on the BY variable.
- In the master data set, each observation must have a unique value for the BY variable.

Example

Create the data sets for this example:

```
libname mylib 'C:\mylib';

data mylib.empmaster;
   set sasuser.payrollmaster;
run;

data mylib.empchanges;
   set sasuser.payrollchanges;
run;
```

Remember that you want to update the master data set *Mylib.Empmaster* with the transactional data set *Mylib.Empchanges*. You can use the UPDATE statement to accomplish this task, as shown the program below. Remember, both data sets must be sorted by or indexed on the BY variable.

```
proc sort data=mylib.empmaster;
   by empid;
run;

proc sort data=mylib.empchanges;
   by empid;
run;

data mylib.empmaster;
   update mylib.empmaster mylib.empchanges;
   by empid;
run;
```

The first eight observations of the updated *Mylib.Empmaster* data set are shown below. Notice that the observation that

has a value of *1065* for **EmpID** now contains the updated values for **JobCode** and **Salary.**

```
proc print data=mylib.empmaster (obs=8) noobs;
run;
```

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
05MAR1957	30MAR1990	1009	M	TA1	$40,432
01 JAN 1956	20OCT1979	1017	M	TA3	$57,201
23MAY1963	27OCT1982	1036	F	TA3	$55,149
14APR1962	17SEP1990	1037	F	TA1	$39,981
13NOV1967	26NOV1989	1038	F	TA1	$37,146
17JUL1961	27AUG1984	1050	M	ME2	$49,234
29JAN1942	10JAN1985	1065	M	ME3	$53,326
180CT1970	06OCT1989	1076	M	PT1	$93,181

Summary

Contents

This section contains the following topics.

Text Summary

Reviewing Terminology

You can review definitions of terms that are important in this chapter. You can also review diagrams and descriptions of the various relationships between input sources for a table lookup operation.

Working with Lookup Values Outside of SAS Data Sets

You can use the IF-THEN/ELSE statement in the DATA step to combine data from a base table with lookup values that are not stored in a SAS data set. You can also use the FORMAT procedure or the ARRAY statement to combine data from a base table with lookup values that are not stored in a SAS data set.

Combining Data with the DATA Step Match-Merge

You can use the MERGE statement in the DATA step to combine data from multiple data sets as long as the input data sets have a common variable. You can merge more than two data sets that lack a common variable in multiple DATA steps if each input data set contains at least one variable in it that is also in at least one other input data set.

Using PROC SQL to Join Data

You can also use PROC SQL to join data from multiple data sets if there is no single variable that is common to all input data sets. In a PROC SQL step, you can choose only the specific variables from each input data set that you want to include in the new data set. If you create a new table with the results of an inner join in a PROC SQL step, the results can be very similar to the results of a DATA step match-merge.

Comparing DATA Step Match-Merges and PROC SQL Joins

It is possible to create identical results with a basic DATA step match-merge and a PROC SQL join. However, there are significant differences between these two methods, as well as advantages and disadvantages to each. In some cases, such as when there is a one-to-one or a one-to-many match on values of the BY variables in the input data sets, these two methods produce identical results. In other cases, such as when there is a many-to-many match on values of the BY

variables, or if there are nonmatching values of the BY variables, these two methods will produce different results. These differences reflect the fact that the processing is different for a DATA step match-merge and a PROC SQL join. Even if you are working with many-to-many matches or nonmatching data, it is possible to use other DATA step techniques such as multiple SET statements to create results that are identical to the results that a PROC SQL step creates.

Combining Summary Data and Detail Data

In order to perform tasks such as calculating percentages based on individual values from a data set as compared to a summary statistic of the data, you need to combine summary data and detail data. One way to create a summary data set is to use PROC MEANS. Once you have a summary data set, you can use multiple SET statements to combine the summary data with the detail data in the original data set. It is also possible to create summary data with a sum statement and to combine it with detail data in one DATA step.

Using an Index to Combine Data

You can use an index to combine data from matching observations in multiple data sets if the index is built on variables that are common to all input data sets. Especially if one of the input data sets is very large, an index can improve the efficiency of the merge. You use the KEY= option in a SET statement in conjunction with another SET statement to use an index to combine data. However, this method might result in data errors in the output data set. You can use the _IORC_ variable to prevent unmatched data from being included in the output data set.

Using a Transactional Data Set

Sometimes, you might want to update the data in one data set with data that is stored in another data set. You use the UPDATE statement to update a master data set with a transactional data set. The UPDATE statement replaces values in the master data set with values from the transactional data set for each observations with a matching value of the BY variable.

Syntax

```
PROC MEANS DATA=original-SAS-data-set NOPRINT;
     <VARvariable(s) ;>
     OUTPUT OUT=output-SAS-data-set statistic=output-variable(s);
RUN;
DATA libref.data-set-name;
     SET SAS-data-set-name;
     SET SAS-data-set-name KEY= index-name;
     variable+expression;
RUN;
DATA master-data-set;
        UPDATE master-data-set transaction-data-set;
        BY by-variables;
RUN;
```

Sample Programs

Combining Data with the IF-THEN/ELSE Statement

```
data mylib.employees_new;
   set mylib.employees;
   if IDnum=10 01 then Birthdate='01JAN1963'd;
   else if IDnum=10 02 then Birthdate='08AUG1946'd;
   else if IDnum=10 03 then Birthdate='23MAR1950'd;
   else if IDnum=10 04 then Birthdate='17JUN1973'd;
run;
```

Combining Data with the ARRAY Statement

```
data mylib.employees_new;
   array birthdates{1001:1004} _temporary_ ('01JAN1963'd
         '08AUG1946'd '23MAR1950'd '17JUN1973'd);
   set mylib.employees;
   Birthdate=birthdates(IDnum);
run;
```

Combining Data with the FORMAT Procedure

```
proc format;
   value $birthdate '1001' = '01JAN1963'
```

```
                    '1002' = '08AUG1946'
                    '1003' = '23MAR1950'
                    '1004' = '17JUN1973';
run;

data mylib.employees_new;
   set mylib.employees;
   Birthdate=input(put(IDnum,$birthdate.),date9.);
run;
```

Performing a DATA Step Match-Merge

```
proc sort data=sasuser.expenses out=expenses;
   by flightid date;
run;

proc sort data=sasuser.revenue out=revenue;
   by flightid date;
run;

data revexpns (drop=rev1st revbusiness revecon
     expenses);
   merge expenses(in=e) revenue(in=r);
   by flightid date;
   if e and r;
   Profit=sum(rev1st, revbusiness, revecon,
          -expenses);
run;

proc sort data=revexpns;
   by dest;
run;

proc sort data=sasuser.acities out=acities;
   by code;
run;

data sasuser.alldata;
   merge revexpns(in=r) acities
         (in=a rename=(code=dest)
         keep=city name code);
   by dest;
   if r and a;
run;
```

Performing a PROC SQL Join

```
proc sql;
   create table sqljoin as
   select revenue.flightid,
          revenue.date format=date9.,
          revenue.origin, revenue.dest,
          sum(revenue.rev1st,
              revenue.revbusiness,
              revenue.revecon)
          -expenses.expenses as Profit,
          acities.city, acities.name
   from sasuser.expenses, sasuser.revenue,
        sasuser.acities
   where expenses.flightid=revenue.flightid
         and expenses.date=revenue.date
         and acities.code=revenue.dest
   order by revenue.dest, revenue.flightid,
            revenue.date;
quit;
```

Working with a Many-to-Many Match

```
proc sql;
   create table flightemp as
   select flightschedule.*, firstname, lastname
      from sasuser.flightschedule, sasuser.flightattendants
```

```
      where flightschedule.empid=flightattendants.empid;
quit;

data fightemps3(drop=empnum jobcode)
   set sasuser.flightschedule;
   do i=1 to num;
      set sasuser.flightattendants
          (rename=(empid=empnum))
          nob=num point=i;
      if empid=empnum then output;
   end;
run;
```

Combining Summary Data and Detail Data

```
proc means data=sasuser.monthsum noprint;
   var revcargo;
   output out=sasuser.summary sum=Cargosum;
run;

data sasuser.percent1;
   if _n_=1 then set sasuser.summary
                     (keep=cargosum);
   set sasuser.monthsum
       (keep=salemon revcargo);
   PctRev=revcargo/cargosum;
run;

data sasuser.percent2(drop=totalrev);
   if _n_=1 then do until(lastobs);
      set sasuser.monthsum(keep=revcargo)
          end=lastobs;
      totalrev+revcargo;
   end;
   set sasuser.monthsum (keep=salemon revcargo);
   PctRev=revcargo/totalrev;
run;
```

Using an Index to Combine Data

```
data work.profit work.errors;
   set sasuser.dnunder;
   set sasuser.sale2000(keep=routeid
       flightid date rev1st revbusiness
       revecon revcargo)key=flightdate;
   if _iorc_=0 then do;
      Profit=sum(rev1st, revbusiness, revecon,
             revcargo, -expenses);
      output work.profit;
   end;
   else do;
      _error_=0;
      output work.errors;
   end;
run;
```

Using a Transactional Data Set

```
proc sort data=mylib.empmaster;
   by empid;
run;

proc sort data=mylib.empchanges;
   by empid;
run;

data mylib.empmaster;
   update mylib.empmaster mylib.empchanges;
   by empid;
run;
```

Points to Remember

- In a DATA step match-merge, you can use the RENAME= option to give identical names to variables in input data sets if those variables record the same information in values that have the same type and length.
- You use the OUTPUT statement and the NOPRINT option with the MEANS procedure if you want the results to be routed to an output data set and the default report to be suppressed.
- The automatic variable _N_ keeps track of how many times a DATA step has iterated. The _N_ variable is useful when you are combining data from a summary data set with data from a larger detail data set.
- When you use the UPDATE statement, both data sets must be sorted by or have indexes based on the BY variable.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. According to the data set descriptions below, which of the variables listed qualify as BY variables for a DATA step match-merge? ?

Variable	Type	Length	Description
Code	char	5	Department code
Totem ps	num	3	Total number of employees
Region	char	4	Location of the department
Manager	num	5	Employee ID number

Variable	Type	Length	Description
IDnum	num	5	Employee ID number
Name	char	20	Employee name
Division	char	3	Division abbreviation
Hiredate	num	8	Date of hire
Supervisor	char	20	Name of supervisor

a. **`Code`** and **`IDnum`**

b. **`Manager`** and **`Supervisor`**

c. **`Manager`** and **`IDnum`**

d. There are no variables that are common to both of these data sets.

2. Suppose you want to merge *Dataset1, Dataset2*, and *Dataset3*. Also suppose that *Dataset1* and *Dataset2* have the common variable **`Startdate`**, *Dataset2* and *Dataset3* have the common variable **`Instructor`**, and that these data sets have no other common variables. How can you use a DATA step to merge these three data sets into one new data set? ?

a. You use a MERGE statement in one DATA step to merge *Dataset1, Dataset2, wadDataset3* by **`Startdate`** and **`Instructor.`**

b. You sort *Dataset1* and *Dataset2* by **`Startdate`** and merge them into a temporary data set in a DATA step. Then you sort the temporary data set and *Dataset3* by **`Instructor`** and merge them into a new data set in a DATA step.

c. You can merge these data sets only with a PROC SQL step.

d. You cannot merge these three data sets at all because they do not have a common variable.

3. Which of the following programs correctly creates a table with the results of a PROC SQL inner join matched on the values of **`empcode?`** ?

a.
```
a. proc sql;
       select newsals.empcode allemps.lastname
```

```
            newsals.salary contrib.amount
        from sasuser.allemps, sasuser.contrib,
            sasuser.newsals
        where empcode=allemps.empid
             and empcode=contrib.empid;
    quit;
```

b. b. proc sql;

```
      create table usesql as
         select newsals.empcode allemps.lastname
                newsals.salsry contrib.amount
                from sasuser.allemps, sasuser.contrib,
                     sasuser.newsals
    quit;
```

c. c. proc sql;

```
      create table usesql as;
         select newsals.empcode, allemps.lastname,
                newsals.salary, contrib.amount;
         from sasuser.allemps, sasuser.contrib,
              sasuser.newsals;
         where empcode=allemps.empid
               and empcode=contrib.empid;
    quit;
```

d. d. proc sql;

```
      create table usesql as
         select newsals.empcode, allemps.lastname,
                newsals.salary, contrib.amount
           from sasuser.allemps, sasuser.contrib,
                sasuser.newsals
           where empcode=allemps.empid
                 and empcode=contrib.empid;
    quit;
```

4. To process a default DATA step match-merge, SAS first reads the descriptor portion of each data set and sets up the PDV and the descriptor portion of the new data set. Which of the following accurately describes the rest of this process? ?

a. Next, SAS sequentially match-merges observations and writes the new observation to the PDV, then to the new data set. When the BY value changes in all the input data sets, the PDV is initialized to missing. Missing values for variables, as well as missing values that result from unmatched observations, are written to the new data set.

b. Next, SAS sequentially match-merges observations and writes the new observation to the PDV, then to the new data set. After each DATA step iteration, the PDV is initialized to missing. Missing values for variables, as well as missing values that result from unmatched observations, are omitted from the new data set.

c. Next, SAS creates a Cartesian product of all possible combinations of observations and writes them to the PDV, then to the new data set. Then SAS goes through the new data set and eliminates all observations that do not have matching values of the BY variable.

d. Next, SAS creates a Cartesian product of all possible combinations of observations and writes them to the PDV, then to the new data set. The new data set is then ordered by values of the BY variable.

5. Which of the following statements is false about using multiple SET statements in one DATA step? ?

a. You can use multiple SET statements to combine observations from several SAS data sets.

b. Processing stops when SAS encounters the end-of-file (EOF) marker on either data set (even if there is more data in the other data set).

c. You can use multiple SET statements in one DATA step only if the data sets in each SET statement have a common variable.

d. The variables in the PDV are not reinitialized when a second SET statement is executed.

6. Select the program that correctly creates a new data set named *Sasuser.Summary* that contains one observation with summary data created from the **Salary** variable of the *Sasuser.Empdata* data set. ?

a.
```
a. proc sum data=sasuser.emdata noprint;
     output out=sasuser.summary sum=Salarysum;
   run;
```

b.
```
b. proc means data=sasuser.empdata noprint;
     var salary;
     output out=sasuser.summary sum=Salarysum;
   run;
```

c.
```
c. proc sum data=sasuser.empdata noprint;
     var salary;
     output out=sasuser.summary sum=Salarysum;
   run;
```

d.
```
d. proc means data=sasuser.empdata noprint;
     output=sasuser.summary sum=Salarysum;
   run;
```

7. If the value of **Cargosum** is *$1000* at the end of the first iteration of the DATA step shown below, what is the value of **Cargosum** in the PDV when the DATA step is in its third iteration? ?

```
data sasuser.percent1;
   if _n_=1 then set sasuser.summary (keep=cargosum);
   set sasuser.monthsum (keep=salemon revcargo);
   PctRev=revcargo/cargosum;
run;
```

a. *$1000*

b. *$3000*

c. The value is missing.

d. The value cannot be determined without seeing the data that is in *Sasuser.Summary*.

8. According to the data set shown, what is the value of **Totalrev** in the PDV at the end of the fourth iteration of the DATA step? ?

```
data sasuser.percent2(drop=totalrev);
   if _n_=1 then do until(lastobs);
      set sasuser.monthsum2(keep=revcargo)
          end=lastobs;
      totalrev+revcargo;
   end;
   set sasuser.monthsum2
       (keep=salemon revcargo);
   PctRev=revcargo/totalrev;
run;
```

Obs	SaleMon	RevCargo
1	JAN1997	$520.00
2	JAN1998	$230.00
3	JAN1999	$350.00
4	FEB1997	.

a. The value is missing.

b. *$350.00*

c. *$520.00*

d. *$1100.00*

9. Which of the following programs correctly uses an index to combine data from two input data sets? ?

a. a.
```
data work.profit;
  set sasuser.sale2 0 00(keep=routeid flightid date
      rev1st revbusiness revecon revcargo)
      key=flightdate;
  set sasuser.dnunder;
  Profit=sum(rev1st, revbusiness, revecon, revcargo,
         -expenses);
run;
```

b. b.
```
data work.profit;
  set sasuser.dnunder;
  set sasuser.sale2 0 00(keep=routeid flightid date
      rev1st revbusiness revecon revcargo)
      key=flightdate;
  where routeid='0000103';
  Profit=sum(rev1st, revbusiness, revecon, revcargo,
         -expenses);
run;
```

c. c.
```
data work.profit;
  set sasuser.dnunder;
  set sasuser.sale2 0 00(keep=routeid flightid date
      rev1st revbusiness revecon revcargo);
      key=flightdate;
  Profit=sum(rev1st, revbusiness, revecon, revcargo,
         -expenses);
run;
```

d. d.
```
data work.profit;
  set sasuser.dnunder;
  set sasuser.sale2 0 00(keep=routeid flightid date
      rev1st revbusiness revecon revcargo)
      key=flightdate;
  Profit=sum(rev1st, revbusiness, revecon, revcargo,
         -expenses);
run;
```

10. Which of the following statements about the _**IORC**_ variable is false? ?

a. It is automatically created when you use either a SET statement with the KEY= option or the MODIFY statement with the KEY= option in a DATA step.

b. A value of zero for _**IORC**_ means that the most recent SET statement with the KEY= option (or MODIFY statement with the KEY= option) did not execute successfully.

c. A value of zero for _**IORC**_ means that the most recent SET statement with the KEY= option (or MODIFY statement with the KEY= option) executed successfully.

d. You can use the _**IORC**_ variable to prevent nonmatching data from being included when you use an index to combine data from multiple data sets.

Answers

1. Correct answer: c

Remember that common variables might not have the same names. **`Manager`** and **`IDnum`** are the only two variables listed that match according to type and description. You can use the RENAME= option to rename one of these variables so that they can be used as BY variables in the MERGE statement of the DATA ste

2. Correct answer: b

In order to merge multiple data sets in a DATA step, the data sets must have a common variable. However, if there are variables that are common to at least two of the input data sets, and if each input data set contains at least one of these variables, then you can use subsequent DATA steps to merge the data sets. You can also use a PROC SQL step to merge data sets that do not have common variables.

3. Correct answer: d

You can use PROC SQL to join data from data sets that do not have a single common variable among them. If you create a new table with the result of an inner join in a PROC SQL step, the resulting data set can be similar or identical to the result of a DATA step match-merge.

4. Correct answer: a

In a DATA step match-merge, SAS reads observations from the input data sets sequentially and match-merges them with observations from other input data sets. Combined observations are created when SAS writes values from all input data sets to the variables in the PDV. These observations, as well as any observations that contain missing or nonmatched values, are then written to the new data set. A PROC SQL join creates a Cartesian product of matches and then eliminates nonmatching data.

5. Correct answer: c

You can use multiple SET statements in one DATA step to combine observations from several data sets, and the data sets do not need to have a common variable. When you use multiple SET statements, you need to keep in mind the process that SAS uses to combine data from the input data sets. Otherwise, you might achieve unexpected results.

6. Correct answer: b

You can use the MEANS procedure to create a new data set that contains a summaiy statistic. You use the NOPRINT option to suppress the default report and the OUTPUT statement to route the results from the MEANS procedure to a new data set. You use the VAR statement to focus the procedure on one or more specific variables from the input data set.

7. Correct answer: a

The **_N_** variable records how many times the DATA step has iterated. In the example shown above, **_N_** is used to ensure that only the first observation is read from Sasuser.Summaiy. Since the values in the PDV are not reinitialized after each DATA step iteration, this value will be retained as long as the DATA step continues to iterate. Therefore, if the value of **Cargosum** is *$1000* in the first iteration, it will be *$1000* in each subsequent iteration as well.

8. Correct answer: d

Totalrev is the accumulator variable of the sum statement, which is automatically initialized with a value of 0. If the expression in a sum statement produces a missing value, SAS replaces the missing value with a value of *0*. As the DATA step iterates, the sum statement retains the accumulator variable so that it will accumulate a total.

9. Correct answer: d

You use the KEY= option in a SET statement to cause SAS to use an index to combine data from multiple data sets. When the SET statement with the KEY= option executes, the program data vector must already contain a value for the indexed variable. You cannot use WHERE processing on a data set that has been read with the KEY= option within the same DATA step.

10. Correct answer: b

When you use the KEY= option, SAS creates an automatic variable named **_IORC_**, which stands for INPUT/OUTPUT Return Code. If the value of **_IORC_** is zero, the index search was successful. The **_IORC_** variable is also created automatically when you use a MODIFY statement in a DATA step.